Διαδικαστικός Προγραμματισμός
(Η γλώσσα C)

Πάρις Μαστοροκώστας

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr

HEALLINK
Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

ΕΣΠΑ 2007-2013
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

ΠΑΡΙΣ ΜΑΣΤΟΡΟΚΩΣΤΑΣ
Καθηγητής Τ.Ε.Ι. Κεντρικής Μακεδονίας

Διαδικαστικός Προγραμματισμός

Η γλώσσα C

Διαδικαστικός Προγραμματισμός (η γλώσσα C)

Συγγραφή

Πάρις Μαστοροκώστας

Κριτικός αναγνώστης

Κλεάνθης Θραμπουλίδης

Συντελεστές έκδοσης

Γλωσσική επιμέλεια: Άννα Μπίσμπα

ISBN: 978-960-603-057-4

ΣΥΝΔΕΣΜΟΣ ΕΛΛΗΝΙΚΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΒΙΒΛΙΟΘΗΚΩΝ

Εθνικό Μετσόβιο Πολυτεχνείο

Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου

www.kallipos.gr

Αφιερώνεται στη Γεωργία, τον Αστέρη και την Ιωάννα-Ραφαέλα,
για την αμέριστη συμπαράσταση και την υπομονή τους

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

Συντομεύσεις – ακρωνύμια

ASCII	American Standard Code for Information Interchange
ANSI	American National Standards Institute
CPL	Cambridge Programming Language
EOF	End of File
FIFO	First In First Out
LIFO	Last In First Out
MS–DOS	Microsoft – Disk Operating System
NAN	Not a Number
RPN	Reverse Polish Notation
stderr	Standard Error
stdin	Standard Input
stdout	Standard Output
VLA	Variable Length Array

1. Εισαγωγή – βασικά στοιχεία προγράμματος

Σύνοψη
Στο κεφάλαιο αυτό αρχικά γίνεται μία εισαγωγή στην έννοια του προγραμματισμού με γλώσσα υψηλού επιπέδου. Παρουσιάζεται η ιστορική εξέλιξη της γλώσσας C και δίνονται τα γενικά χαρακτηριστικά της γλώσσας. Ακολουθεί η περιγραφή των εργαλείων ανάλυσης προβλημάτων προγραμματισμού και των σταδίων υλοποίησης προγράμματος. Στη συνέχεια μελετώνται τα βασικά στοιχεία των προγραμμάτων και το λεξιλόγιο της C και το κεφάλαιο ολοκληρώνεται με την αποτύπωση κανόνων για τη δημιουργία ευανάγνωστων προγραμμάτων.

Λέξεις κλειδιά
γλώσσα προγραμματισμού υψηλού επιπέδου, διαδικαστικός προγραμματισμός, γλώσσα C, πρότυπο γλώσσας, φυσική γλώσσα, διάγραμμα ροής, ψευδοκώδικας, συντάκτης, μεταγλωττιστής, συνδέτης, εκτελέσιμο αρχείο, πηγαίος κώδικας, αντικείμενο αρχείο, συντακτικά και σημασιολογικά σφάλματα, σχόλια, λεξεις κλειδιά, δεσμευμένες λέξεις, αναγνωριστές, main, αρχείο κεφαλίδας

1.1. Εισαγωγή

Αντικείμενο του παρόντος συγγράμματος είναι η εισαγωγή του αναγνώστη στη λογική του προγραμματισμού Η/Υ. Το ενδιαφέρον εστιάζεται στον επονομαζόμενο **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 προχωρήσει σε άλλες μορφές προγραμματισμού, όπως ο αντικειμενοστραφής προγραμματισμός (object–oriented programming), έχοντας αποκτήσει τις γνώσεις που αποτελούν κοινό τόπο ανάμεσα στα είδη προγραμματισμού.

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 διάφορων 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 ως επί το πλείστον *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.
- Έχει φορητότητα, δηλαδή ο κώδικάς της μεταγλωττίζεται από διάφορους μεταγλωττιστές χωρίς να απαιτούνται τροποποιήσεις.
- Μετά από τέσσερις και πλέον δεκαετίες συνεχίζει να αποτελεί μία από τις ευρύτερα χρησιμοποιούμενες γλώσσες προγραμματισμού, γεγονός που έχει δημιουργήσει πολύ μεγάλη εγκατεστημένη βάση εφαρμογών που αναπτύχθηκαν με αυτήν τη γλώσσα και πρέπει να συντηρούνται και να εξελίσσονται.
- Αποτελεί μία εξαιρετική εκκίνηση για την εκμάθηση γλωσσών που στηρίζονται στον αντικειμενοστραφή προγραμματισμό (C++, Java, C#), καθώς ο τελευταίος έχει δανειστεί πολλά χαρακτηριστικά από τη C.

Ωστόσο, η γλώσσα C παρουσιάζει και μία σειρά μειονεκτημάτων, που την καθιστούν απαιτητική στον χειρισμό. Παρέχοντας μεγάλο βαθμό ελευθερίας στον προγραμματιστή και χαρακτηριζόμενη από έλλειψη περιορισμών και μικρό βαθμό ελέγχου λαθών, υποχρεώνει τον προγραμματιστή να είναι ιδιαίτερα προσεκτικός και να χρησιμοποιεί διαδικασίες ελέγχου λαθών, οι οποίες παρέχονται αυτόματα σε άλλες γλώσσες προγραμματισμού. Επιπλέον, σε πολλές περιπτώσεις εισάγονται λάθη που είναι μη ανιχνεύσιμα από τον μεταγλωττιστή και οδηγούν σε καταστάσεις απροσδιοριστίας, δηλαδή μη αναμενόμενες καταστάσεις με μη προσδιορισμένη συμπεριφορά.

1.2. Ιστορική αναδρομή της γλώσσας C

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 στο πλαίσιο του προγράμματος MAC και του απογόνου 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, αποτέλεσαν τη θερμοκοιτίδα πολλών προγραμμάτων λογισμικού, που κυριαρχούν από τη δεκαετία του 1960 έως σήμερα.

Η C, ούσα ευέλικτη και αποδοτική, χρησιμοποιήθηκε αρχικά για τον προγραμματισμό συστημάτων στο Unix. Το 1974 εμφανίστηκε 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 αποτέλεσε 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 το 1989 στην επονομαζόμενη **ANSI έκδοση**, που την προτυποποίησε (το όνομά της το έλαβε από την επιτροπή του American National Standards Institute). Μία σημαντική ανανέωση έγινε το 1999 με το πρότυπο C99, ενώ το τελευταίο πρότυπο παρουσιάστηκε το 2011. Ωστόσο, τα καινοτόμα στοιχεία του προτύπου C11, όπως η προτυποποίηση του πολυνηματικού προγραμματισμού, δεν έχουν ακόμη ενσωματωθεί στους περισσότερους μεταγλωττιστές της γλώσσας.

Τις τελευταίες δύο δεκαετίες τα ηνία έχει λάβει ο αντικειμενοστρεφής προγραμματισμός και στην κατεύθυνση αυτή αρχικά συνέβαλε η γλώσσα C++, που επινοήθηκε το 1983 από τον Δανό Bjarne Stroustrup στα εργαστήρια της AT&T (το τμήμα των εργαστηρίων Bell που μεταφέρθηκε στην AT&T, όταν η πρώτη διασπάστηκε). Η C++ – ή *C με τάξεις* – μπορεί να θεωρηθεί απόγονος της C, αν και έχει αρκετές διαφορές. Πάντως, θα πρέπει να σημειωθεί ότι, σύμφωνα με μετρήσεις σχετικών οργανισμών για την επαγγελματική και ακαδημαϊκή χρήση των γλωσσών προγραμματισμού, το 2015 η γλώσσα C συνεχίζει να αποτελεί τη γλώσσα στην οποία έχουν γραφτεί οι περισσότερες γραμμές κώδικα.

1.3. Εργαλεία ανάλυσης προβλημάτων

Για την ανάλυση ενός προβλήματος και την κατάρτιση του μοντέλου που θα υλοποιηθεί σε κώδικα έχουν επικρατήσει τα ακόλουθα τρία εργαλεία:

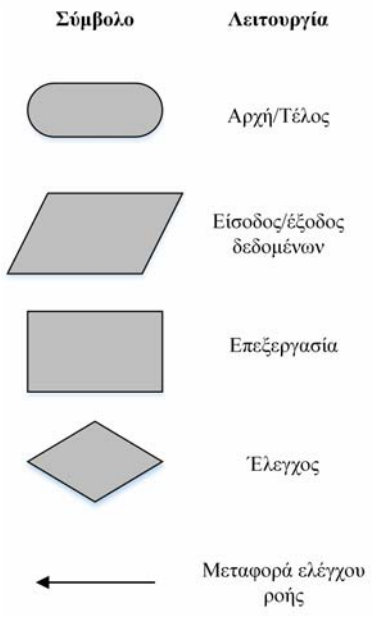
1. Η **φυσική γλώσσα** (natural language), σύμφωνα με την οποία το πρόβλημα αναλύεται σε απλές προτάσεις της καθομιλούμενης γλώσσας, χρησιμοποιώντας το συντακτικό αυτής.

2. Ο ψευδοκώδικας (pseudocode), ο οποίος μετασχηματίζει τη φυσική γλώσσα σε μία σειρά προτάσεων που χρησιμοποιούν το συντακτικό γλώσσας προγραμματισμού, χωρίς να ακολουθούν επακριβώς τον φορμαλισμό συγκεκριμένης γλώσσας.

3. Το λογικό διάγραμμα ή διάγραμμα ροής (flow chart), σύμφωνα με το οποίο απεικονίζεται γραφικά η λύση του προβλήματος, με χρήση ειδικών συμβόλων.

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. Το λογικό διάγραμμα χρησιμοποιείται ευρύτατα, όταν συνεργάζονται ομάδες προγραμματιστών, γιατί παρέχει μία σειρά από πλεονεκτήματα, καθώς προσδίδει μεγαλύτερη ευχέρεια *(α)* στην παραστατική ανάλυση του προβλήματος, *(β)* στην τεκμηρίωση του προγράμματος, *(γ)* στην αποδοτική συντήρηση του προγράμματος. Ωστόσο, έχει περιορισμούς, καθόσον *(α)* σύνθετα προγράμματα οδηγούν σε σύνθετα και δύσχρηστα διαγράμματα ροής και *(β)* τροποποιήσεις στα προγράμματα μπορεί να απαιτήσουν επανασχεδιασμό ολόκληρου του διαγράμματος.

Τα σύμβολα που χρησιμοποιούνται στο διάγραμμα ροής παρουσιάζονται στο **Σχήμα 1.1**:



Σχήμα 1.1 *Τα σύμβολα του διαγράμματος ροής*

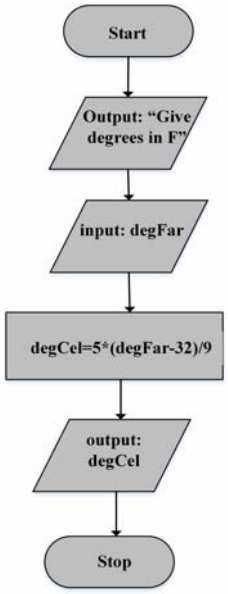
1.3.1. Παράδειγμα

Μία ενδεικτική περιγραφή του τρόπου εφαρμογής των ανωτέρω εργαλείων θα γίνει με τη βοήθεια του ακόλουθου προβλήματος: Να μετατραπούν οι βαθμοί Farenheit σε βαθμούς Κελσίου, χρησιμοποιώντας την εξίσωση μετασχηματισμού $C = 5 \cdot (F - 32) / 9$.

1. Με χρήση φυσικής γλώσσας

Ζήτησε από τον χρήστη τη θερμοκρασία σε βαθμούς F
Διάβασε την τιμή που δίνει ο χρήστης
Αποθήκευσε την τιμή σε θέση αποθήκευσης που ονομάζεται `degFar`
Υπολόγισε τους βαθμούς C με χρήση μαθηματικής σχέσης
Αποθήκευσε το αποτέλεσμα σε θέση αποθήκευσης που ονομάζεται `degCel`
Τύπωσε το περιεχόμενο της `degCel`

2. Με χρήση διαγράμματος ροής



Σχήμα 1.2 *Το διάγραμμα ροής του παραδείγματος 1.3.1*

3. Με χρήση ψευδοκώδικα

```
print "enter degrees in Farenheit"
read degFar
degCel = (degFar - 32) * 5 / 9
print degCel
```

1.4. Στάδια ανάπτυξης 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– ο επονομαζόμενος και *κύκλος δημιουργίας προγράμματος*– περιλαμβάνει τέσσερα στάδια:

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

2. Η μεταγλώττιση του πηγαίου κώδικα (compilation). Η διαδικασία της μεταγλώττισης εκτελείται από τον **μεταγλωττιστή** (compiler) και παράγεται ένα αρχείο που ονομάζεται **αρχείο αντικειμένου** (object file) και περιέχει τον κώδικα 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 εκτελεστεί το πρόγραμμα). Εάν δεν υπάρχουν συντακτικά σφάλματα, το αρχείο που προκύπτει έχει το όνομα του αρχείου του πηγαίου κώδικα και κατάληξη **.obj**.

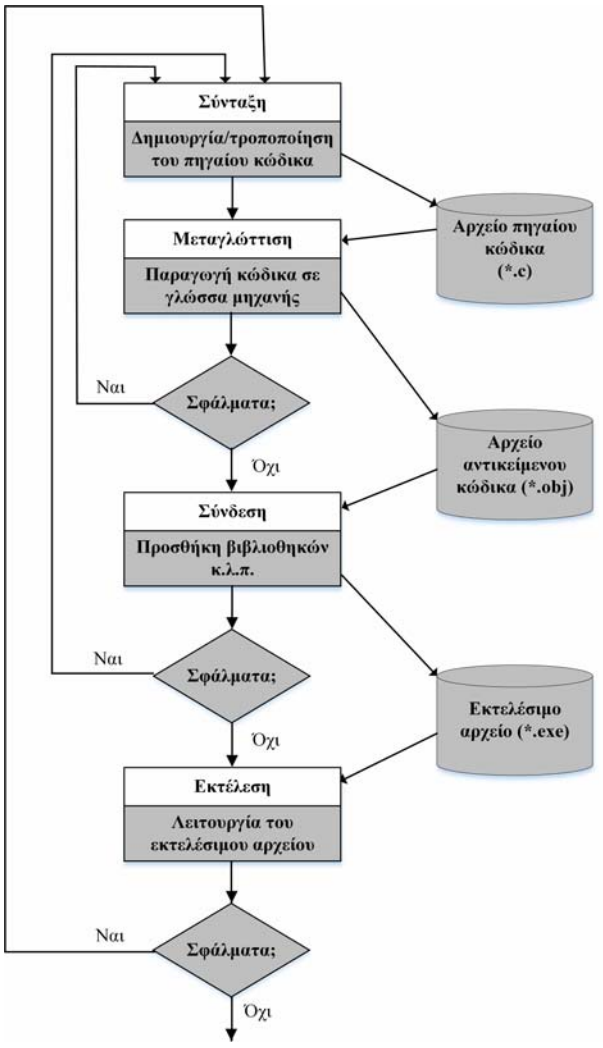
3. Η σύνδεση του αντικειμένου με βιβλιοθήκες. Στο τρίτο στάδιο επιτελείται η διεργασία της σύνδεσης (linking), η οποία είναι πλήρως αυτοματοποιημένη και γίνεται με χρήση του **συνδέτη** (linker). Το αποτέλεσμα της σύνδεσης είναι η δημιουργία του εκτελέσιμου κώδικα ή η αναφορά τυχόν προβλημάτων, όπως 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 μπορεί να εκτελεστεί όπως ένα οποιοδήποτε εκτελέσιμο αρχείο του υπολογιστή.

Ο συνδέτης έχει τη δυνατότητα να συνδέσει περισσότερα του ενός αρχεία αντικειμένου. 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 στον 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.

Στο **Σχήμα 1.3** αποτυπώνονται εποπτικά τα στάδια υλοποίησης προγράμματος:



Σχήμα 1.3 *Τα στάδια υλοποίησης προγράμματος*

1.5. Βασικά στοιχεία προγράμματος

Έστω το ακόλουθο απλό πρόγραμμα:

```
/*****************************************************
 This program prints out the sentence "This is a test."
 *****************************************************/

#include <stdio.h>

int main()
{
  printf( "This is a test.\n" );
  return 0;
}  // End of main
```

Η καρδιά ενός προγράμματος της γλώσσας C είναι η λέξη **main**. Αντιστοιχεί στο κύριο τμήμα του κώδικα (κύρια *συνάρτηση* του προγράμματος, γι' αυτό και εφεξής θα αναφέρεται ως **main()**) και χρησιμοποιείται για να γνωστοποιήσει το σημείο έναρξης της εκτέλεσης του προγράμματος. Μετά τη **main()** ακολουθεί το εισαγωγικό αριστερό άγκιστρο (**{**) και κατόπιν οι προτάσεις του προγράμματος. Η **main()** και μαζί το πρόγραμμα τερματίζουν με το καταληκτικό δεξί άγκιστρο (**}**). Στο παρόν πρόγραμμα η **main()** αποτελείται από μία πρόταση κλήσης της συνάρτησης **printf()**, η οποία ανήκει στη βασική βιβλιοθήκη συναρτήσεων της C, καθώς και από την πρότυπη τελευταία πρόταση των προγραμμάτων της γλώσσας C **return 0;**. 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* (preprocessor directive) **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 τον 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. Λεπτομερής ανάλυση του συντακτικού και της λειτουργίας των συναρτήσεων θα δοθεί στο Κεφάλαιο 4.

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 εμφάνιση στην οθόνη της πρότασης: **This is a test.**

Οι τρεις πρώτες γραμμές του ανωτέρω προγράμματος είναι **σχόλια** (comments) και δεν αποτελούν τμήμα του κώδικα, καθώς δεν λαμβάνονται υπόψη από τον 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 καθιστούν 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:

(α)
```
/* Αυτό /* το σχόλιο */ είναι λανθασμένο */
```
(β)
```
/*
      Αυτό το
        σχόλιο
είναι σωστό
        */
```

Παρατηρήσεις:

1. Στο παρόν σύγγραμμα ο όρος **εντολή** (command) αφορά *σε κάθε συντακτική οντότητα που είτε μεταβάλλει κάποια τιμή είτε επιτρέπει τη ροή πληροφορίας από και προς το εξωτερικό περιβάλλον*. Υπ' αυτήν την έννοια, οι συναρτήσεις θεωρούνται εντολές. Έτσι, π.χ. η συνάρτηση **`printf`** είναι μία εντολή, καθώς μεταφέρει πληροφορία στο εξωτερικό περιβάλλον.

2. 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!

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

4. Η συνάρτηση **`printf()`** ανήκει στις **μορφοποιούμενες** συναρτήσεις εισόδου– εξόδου. Ονομάζεται μορφοποιούμενη, γιατί δίνει τη δυνατότητα στον 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.

5. Πέραν της **`include`**, μία σημαντική οδηγία προς τον προεπεξεργαστή είναι η **`define`**, η οποία αντιστοιχίζει ένα όνομα σε μία σταθερά ή σε μία σειρά χαρακτήρων (εκτελεί *μακροαντικατάσταση*). 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 εμφανιστεί στην οθόνη με χρήση της **`printf`**:

```
#define TITLOS  "C Programming \ne-book\n"
printf( TITLOS );
```

Το αποτέλεσμα είναι:

```
C Programming
e-book
```

Μία συνηθισμένη χρήση της **`define`** είναι για τον καθορισμό του μεγέθους στοιχείων, όπως είναι η διάσταση ενός πίνακα, τα οποία μπορεί να χρησιμοποιηθούν επανειλημμένα μέσα στον κώδικα. Επιπρόσθετα, μέσω της **`define`** μπορούν να οριστούν εντολές, που ονομάζονται **μακροεντολές**. Τέλος, όπως θα αναλυθεί στο Κεφάλαιο 10, η **`define`** εμπλέκεται στην επονομαζόμενη *υπό συνθήκη μεταγλώττιση*.

1.6. Λεξιλόγιο της γλώσσας C

Από τα αναφερθέντα στην προηγούμενη παράγραφο γίνεται φανερό ότι η λειτουργία της γλώσσας 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 γλώσσα 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.λπ.
- Μακροονόματα (macro names). Είναι ονόματα που περιέχονται σε αρχεία κεφαλίδας για ορισμό μακροεντολών, π.χ. **`EOF`**, **`INT_MAX`**.
- Ονόματα τύπων (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 τον 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 1.1** παρατίθενται οι λέξεις κλειδιά της γλώσσας C:

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

Πίνακας 1.1 *Λέξεις κλειδιά της γλώσσας 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 στο Κεφάλαιο 2.

1.7. Κανόνες δημιουργίας ευανάγνωστων προγραμμάτων

- Θα πρέπει να αποφεύγονται ονόματα ενός χαρακτήρα, όπως **`i`**, **`j`**, **`x`**, **`y`** (εκτός από ειδικές περιπτώσεις που θα αναφερθούν στα επόμενα κεφάλαια).

- Το όνομα που χρησιμοποιείται θα πρέπει να είναι ενδεικτικό του τι αναπαριστά ή διαχειρίζεται η μεταβλητή. Συγκεκριμένα:

    1. Η μεταβλητή που αναπαριστά τον όγκο θα μπορούσε να ονομαστεί **`volume`** και η μέγιστη τιμή της **`max_volume`** ή **`maxVolume`**.

    2. Η συνάρτηση που εμφανίζει στην οθόνη πληροφορίες για τη λειτουργία του προγράμματος μπορεί να λάβει το ενδεικτικό όνομα **`printProgramInfo`**.

- Σε ολόκληρο τον κώδικα ενός προγράμματος θα πρέπει να διατηρείται ο ίδιος φορμαλισμός στην ονοματοδοσία. Κατά συνέπεια, εάν χρησιμοποιηθεί η σύμβαση σύζευξης δύο ή περισσότερων λέξεων σε ένα όνομα μεταβλητής μέσω του χαρακτήρα υπογράμμισης **`_`** (π.χ. **`string_name`**), δεν θα πρέπει να χρησιμοποιηθεί άλλος τρόπος σύζευξης (π.χ. **`stringName`**) αλλά να διατηρηθεί ο αρχικός (δηλαδή **`string_name)`**.

- Θα πρέπει να χρησιμοποιούνται μικρά γράμματα για ονόματα μεταβλητών και κεφαλαία γράμματα για χρήση συνωνύμων μέσω της εντολής προεπεξεργαστή **`define`**.

Ερωτήσεις αυτοαξιολόγησης

(1) Η γλώσσα C αποτελεί:
(α) Γλώσσα προγραμματισμού υψηλού επιπέδου
(β) Γλώσσα μηχανής
(γ) Γλώσσα assembly
(δ) Τίποτε από τα προηγούμενα

(2) Ποια είναι η σωστή σειρά εκτέλεσης των ακόλουθων βημάτων;
(α) i. Μεταγλώττιση και δημιουργία του αρχείου αντικειμένου
ii. Δημιουργία του εκτελέσιμου κώδικα
iii. Συγγραφή του πηγαίου κώδικα
iv. Προεπεξεργασία και ενσωμάτωση των αρχείων κεφαλίδας
(β) i. Συγγραφή του πηγαίου κώδικα
ii. Προεπεξεργασία και ενσωμάτωση των αρχείων κεφαλίδας
iii. Μεταγλώττιση και δημιουργία του αντικείμενου κώδικα
iv. Δημιουργία του εκτελέσιμου κώδικα
(γ) i. Προεπεξεργασία και ενσωμάτωση των αρχείων κεφαλίδας
ii. Συγγραφή του πηγαίου κώδικα
iii. Μεταγλώττιση και δημιουργία του αρχείου αντικειμένου
iv. Δημιουργία του εκτελέσιμου κώδικα
(δ) i. Μεταγλώττιση και δημιουργία του αρχείου αντικειμένου
ii. Συγγραφή του πηγαίου κώδικα
iii. Προεπεξεργασία και ενσωμάτωση των αρχείων κεφαλίδας
iv. Δημιουργία του εκτελέσιμου κώδικα

(3) Ποιο από τα ακόλουθα σχόλια είναι σωστό;
(α)
```
/* Σχόλιο σχόλιο σχόλιο σχόλιο
```
(β)
```
/* Σχόλιο σχόλιο /* σχόλιο σχόλιο */
```
(γ)
```
/* Σχόλιο
   σχόλιο  σχόλιο
   σχόλιο  */
```
(δ)
```
/* Σχόλιο σχόλιο σχόλιο σχόλιο    //
```

(4) Το όνομα **`typedef`** αποτελεί:
(α) Λέξη κλειδί

(β) Τελεστή
(γ) Αναγνωριστή
(δ) Δεσμευμένη λέξη

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Gabrielli, M. & Martini, S. (2010), *Programming Languages: Principles and Paradigms*, Springer.

Horton, I. (2006), *Beginning C – from Novice to Professional*, 4th ed., Apress.

Kernighan, B. & Ritchie, D. (1990), *Η γλώσσα προγραμματισμού C*, Εκδόσεις Κλειδάριθμος.

Lohr, S. (2001), *Go To*, Basic Books.

2. Μεταβλητές – Είσοδος/έξοδος προγράμματος – Εκφράσεις και Τελεστές

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια και τη χρήση της μεταβλητής στον προγραμματισμό. Αναλύονται τα είδη μεταβλητών στη γλώσσα C, η εσωτερική απεικόνισή τους και τα λειτουργικά χαρακτηριστικά τους. Ακολούθως, περιγράφονται οι βασικές συναρτήσεις ανάγνωσης και εκτύπωσης δεδομένων της γλώσσας C, οι οποίες επιτελούν την επικοινωνία του προγράμματος με το εξωτερικό περιβάλλον (συσκευές εισόδου και εξόδου). Στη συνέχεια, παρουσιάζονται οι κατηγορίες εκφράσεων και μελετώνται τα είδη των τελεστών: αύξησης– μείωσης, ανάθεσης, συσχετιστικοί και λογικοί τελεστές, καθώς και ο τελεστής sizeof. Ακολουθεί η ανάλυση των εννοιών και των ιδιοτήτων της προτεραιότητας και της προσεταιριστικότητας και το κεφάλαιο ολοκληρώνεται με τις διαδικασίες της ρητής και της έμμεσης μετατροπής τύπων.

Λέξεις κλειδιά
μεταβλητή, σταθερά, τύπος χαρακτήρα, τύπος ακεραίου, τύπος αριθμού κινητής υποδιαστολής, εκφράσεις, σημειολογίες τελεστών, αριθμητικοί τελεστές, λογικοί τελεστές, συσχετιστικοί τελεστές, τελεστές αύξησης– μείωσης, προτεραιότητα και προσεταιριστικότητα τελεστών, ρητή– έμμεση μετατροπή τύπου, scanf, printf, puchar, getchar, sizeof

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C– στάδια υλοποίησης προγράμματος

2.1. Η έννοια της μεταβλητής

Ένα από τα πλέον βασικά πλεονεκτήματα του ηλεκτρονικού υπολογιστή είναι η δυνατότητά του να διαχειρίζεται πληροφορία σε μορφή αριθμητικών δεδομένων, γραμμάτων ή ακολουθίας γραμμάτων. Τα δεδομένα αποθηκεύονται στη μνήμη και θα πρέπει να μπορούν να είναι προσβάσιμα 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 διαχειρίζονται δεδομένα. Το όνομα μίας μεταβλητής είναι άμεσα συνδεδεμένο με τη διεύθυνση στην οποία είναι αποθηκευμένο το δεδομένο. Με τον τρόπο αυτό ο προγραμματιστής μπορεί να χειριστεί 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=3*x+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 δεν είναι καθορισμένη, έως ότου χρησιμοποιηθεί για πρώτη φορά από το πρόγραμμα. Η γλώσσα C απαιτεί δήλωση μίας μεταβλητής πριν τη χρήση της, καθότι η δήλωση συνεπάγεται τον προσδιορισμό του τύπου δεδομένου που θα χειρίζεται και, συνακόλουθα, τη μνήμη που θα πρέπει να της ανατεθεί. Συνήθως οι μεταβλητές δηλώνονται στην αρχή της

συνάρτησης, ως επί το πλείστον αμέσως μετά το εισαγωγικό άγκιστρο (`{`). Ωστόσο, το πρότυπο της γλώσσας C99 επιτρέπει τη δήλωση σε οποιοδήποτε σημείο του κώδικα, πάντοτε όμως πριν την πρώτη χρήση τους.

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. Η δήλωση γνωστοποιεί στον μεταγλωττιστή το όνομα της μεταβλητής και τον τύπο δεδομένου που θα χειρίζεται. Μία δήλωση έχει ως αποτέλεσμα τη σύνδεση του ονόματος της μεταβλητής με:

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

2.1.2. Ονομασία μεταβλητής

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

- Στη γλώσσα C τα ονόματα των μεταβλητών σχηματίζονται από:
  (α) τα γράμματα του αλφαβήτου,
  (β) τα ψηφία 0 έως 9,
  (γ) τον 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.
- Τα όνομα μίας μεταβλητής πρέπει να είναι ενδεικτικό της ιδιότητάς της ή του τύπου δεδομένου που αντιπροσωπεύει, έτσι ώστε να δίνει πληροφορία στον προγραμματιστή. Στην περίπτωση που το όνομα μίας μεταβλητής περιέχει περισσότερες από μία λέξεις, συνιστάται το πρώτο γράμμα κάθε λέξης να είναι κεφαλαίο.

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:
  `j  x2  max_number_of_students_in_my_class`

Σημείωση: 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.

2.2. Τύποι μεταβλητών

Οι μεταβλητές της γλώσσας 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 χωρίζονται 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).
- Τον απαριθμητικό τύπο (enumerated, **enum**).

Στο πρότυπο της γλώσσας C99 προστέθηκε και ο τύπος **bool**, ο οποίος εμφανίζει δύο τιμές (**true** που αντιστοιχεί στον ακέραιο **1** και **false** που αντιστοιχεί στον ακέραιο **0**). Επειδή όμως στην πραγματικότητα αυτός ο τύπος είναι τεχνητός, υπό την έννοια ότι τα αποτελέσματα προκύπτουν μέσα από επεξεργασία ακεραίων από μακροεντολές, δεν θα μελετηθεί.

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 στο Κεφάλαιο 5 αναπτύσσονται οι πίνακες. Οι άλλοι τύποι μεταβλητών θα μελετηθούν στο Κεφάλαιο 8.

2.2.1. Ο τύπος του χαρακτήρα

Ο τύπος του χαρακτήρα (**char**) παριστάνει χαρακτήρες του αλφάβητου της γλώσσας. Βρίσκεται ανάμεσα σε 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** **%c** (character). Η πρόταση

```
scanf( "%c", &choice );
```

διαβάζει από την κύρια είσοδο (πληκτρολόγιο) έναν χαρακτήρα και τον αποδίδει στη μεταβλητή **choice**. Θα πρέπει να προσεχθεί η χρήση του **&** πριν από τη μεταβλητή. Ονομάζεται **τελεστής διεύθυνσης** και προηγείται των μεταβλητών στη συνάρτηση **scanf()** (με εξαίρεση τα ονόματα μεταβλητών που αφορούν σε πίνακες, για τους οποίους θα γίνει σχετικός σχολιασμός στο Κεφάλαιο 5).

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);
```

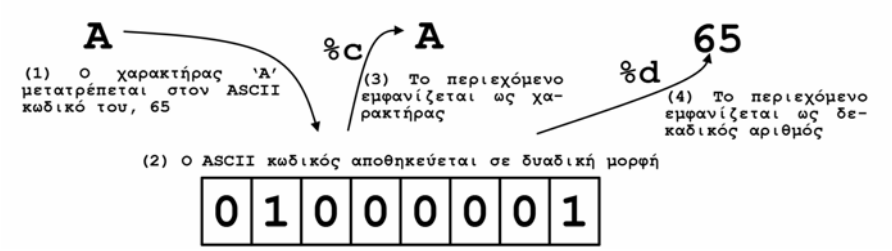
θα τυπώσει

```
The ASCII Code of A is 65
```

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



Σχήμα 2.1 *Αποθήκευση και ανάκληση ASCII χαρακτήρα*

2.2.2. Ο κώδικας (πίνακας) ASCII

Ο κώδικας ASCII αντιστοιχεί χαρακτήρες στις 256 καταστάσεις που μπορεί να περιέχει ένα byte. Η τυπική κωδικοποίηση ASCII χρησιμοποιεί τις πρώτες 128 καταστάσεις, με κωδικούς από το δεκαδικό 0 (δυαδική απεικόνιση σε byte: 00000000) έως το δεκαδικό 127 (δυαδική απεικόνιση σε byte: 01111111). Οι υπόλοιπες 128 καταστάσεις (128 έως 255 ή ισοδύναμα στο δυαδικό σύστημα 10000000 έως 11111111) αποτελούν επέκταση του πρότυπου κώδικα (extended ASCII code) και διαφέρουν ανάλογα με το σύστημα.

Οι πρώτες 31 θέσεις του πίνακα ASCII είναι μη εκτυπούμενοι χαρακτήρες και χρησιμοποιούνται ως χαρακτήρες ελέγχου. Στον **Πίνακα 2.1** απεικονίζεται ο πρότυπος πίνακας ASCII, όπου δίπλα στον κωδικό ASCII εμφανίζεται ο αντίστοιχος χαρακτήρας.

ASCII		ASCII		ASCII		ASCII		ASCII		ASCII		ASCII		ASCII	
0		16		32		48	0	64	@	80	P	96	‘	112	p
1		17		33	!	49	1	65	A	81	Q	97	a	113	q
2		18		34	«	50	2	66	B	82	R	98	b	114	r
3		19		35		51	3	67	C	83	S	99	c	115	s
4		20		36	$	52	4	68	D	84	T	100	d	116	t
5		21		37	%	53	5	69	E	85	U	101	e	117	u
6		22		38	&	54	6	70	F	86	V	102	f	118	v
7		23		39	‘	55	7	71	G	87	W	103	g	119	w
8		24		40	(	56	8	72	H	88	X	104	h	120	x
9		25		41	)	57	9	73	I	89	Y	105	i	121	y
10		26		42	*	58	:	74	J	90	Z	106	j	122	z
11		27		43	+	59	;	75	K	91	[	107	k	123	{
12		28		44	,	60	<	76	L	92	\	108	l	124	|
13		29		45	-	61	=	77	M	93	]	109	m	125	}
14		30		46	.	62	>	78	N	94	^	110	n	126	~
15		31		47	/	63	?	79	O	95	_	111	o	127	

Πίνακας 2.1 *Ο κώδικας ASCII*

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 4 bytes (32 bits) για κάθε ακέραιο, το σύνολο των δυνατών τιμών είναι $2^{32} = 4.284.967.296$. Εάν θεωρηθεί ότι τις

μισές τιμές θα καταλάβουν αρνητικοί ακέραιοι και τις υπόλοιπες μισές θετικοί, η περιοχή τιμών του τύπου **int** είναι από -2.147.483.648 +2.147.483.647, με την ακόλουθη αντιστοιχία δυαδικής και δεκαδικής τιμής:

$(00000000000000000000000000000000)_2 \rightarrow (-2.147.483.648)_{10}$

$(00000000000000000111111111111111)_2 \rightarrow (-1)_{10}$

$(00000000000000001000000000000000)_2 \rightarrow (0)_{10}$

$(11111111111111111111111111111111)_2 \rightarrow (+2.147.483.647)_{10}$

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 τον ίδιο αποθηκευτικό χώρο με τον απλό ακέραιο. Ωστόσο, στο πρότυπο C99 έχει προστεθεί ο τύπος **long long int**, για τις μεταβλητές του οποίου διατίθεται χώρος 64 bits. Επιπλέον, ο προσδιοριστής **unsigned** χρησιμοποιείται πριν από τη λέξη **int**, για να χαρακτηρίσει τη μεταβλητή χωρίς πρόσημο. Το πεδίο τιμών ενός μη προσημασμένου ακέραιου μήκους *n* bits εκτείνεται από 0 έως $2^n - 1$.

Η εισαγωγή τιμών στις ακέραιες μεταβλητές γίνεται με χρήση της συνάρτησης **scanf()** και του προσδιοριστή **%d** (από τη λέξη *decimal*). 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 καθοριστεί 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 δεν θα ληφθεί υπόψη. Στην αντίθετη περίπτωση, στις πλεονάζουσες θέσεις θα τοποθετηθούν κενά. Για τον 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. 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 τον χειρίζεται ως ***ακέραια σταθερά***. Η σταθερά **245** αποθηκεύεται ως **int**, ενώ η σταθερά **100000** αποθηκεύεται ως **long int**. Εάν οριστεί η σταθερά **8965** ως **8965LL**, ο μεταγλωττιστής αναγκάζεται να δεσμεύσει χώρο για **long long int**.

2.2.3.1. Παράδειγμα

Να καταστρωθεί πρόγραμμα που να εξετάζει τα μήκη και τα πεδία τιμών των διάφορων παραλλαγών του τύπου ακεραίου.

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

int main()
{
   /* INT_MAX είναι ο μέγιστος int: ορίζεται στο αρχείο κεφαλίδας
   limits.h. Ο τελεστής sizeof δίνει το μέγεθος ενός τύπου δεδομένου
   ή μίας μεταβλητής */
   printf( "The size of type int is %d bytes\n",sizeof(int) );
   printf( "The size of type short is %d bytes\n",sizeof(short) );
   printf( "The size of type long is %d bytes\n",sizeof(long) );
   printf( "The size of type long long is %d bytes\n",sizeof(long
long) );
   printf( "\nmin int: %d,   max int: %d\n",INT_MIN,INT_MAX );
   printf( "\nmin short: %d,   max short: %d\n",SHRT_MIN,SHRT_MAX );
   printf( "\nmin long: %d,   max long: %d\n",LONG_MIN, LONG_MAX );

   return 0;
}
```

```
The size of type int is 4 bytes
The size of type short is 2 bytes
The size of type long is 4 bytes
The size of type long long is 8 bytes

min int: -2147483648,   max int: 2147483648

min short: -32768,   max short: 32767

min long: -2147483648,   max long: 2147483648
```

Εικόνα 2.1 *Η έξοδος του προγράμματος του παραδείγματος 2.2.3.1*

Από τα αποτελέσματα προκύπτει ότι ταυτίζονται τα bytes για `int` και `long`, ενώ ο πλέον εκτεταμένος τύπος ακεραίου `long long int` καταλαμβάνει 8 bytes.

2.2.3.2. Παράδειγμα

Να καταστρωθεί πρόγραμμα που να επιτελεί τα παρακάτω:

Ζήτησε από τον χρήστη έναν χαρακτήρα
Πάρε από τον χρήστη τον χαρακτήρα
Τύπωσε τον χαρακτήρα και τον ASCII κωδικό του
Βρες τον επόμενο χαρακτήρα
Τύπωσέ τον μαζί με τον κωδικό του

```
#include <stdio.h>
int main()
{
```

```
    char charVar,nextChar;
    printf( "\tWrite a character:\t" );
    scanf( "%c",&charVar );
    printf( "\tThe ASCII code of char %c is %d\n", charVar, charVar );
    nextChar = charVar + 1;      /* βρίσκει τον επόμενο χαρακτήρα */
    printf( "\tThe ASCII code of char %c is %d\n", nextChar, nextChar );

    return 0;
}
```

```
Write a character:      D
The ASCII code of character D is 68
The ASCII code of character E is 69
```

Εικόνα 2.2 *Η έξοδος του προγράμματος του παραδείγματος 2.2.3.2*

2.2.4. Τύποι πραγματικών αριθμών

Οι πραγματικοί αριθμοί είναι οι αριθμοί που διαθέτουν κλασματικό μέρος και εκφράζονται συνήθως στις μορφές του **Πίνακα 2.2**:

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

Πίνακας 2.2 *Σημειογραφίες των πραγματικών αριθμών*

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. Ένας νεότερος τύπος εκτεταμένης ακρίβειας είναι ο **`long double`**.

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

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

Όπως και στις περιπτώσεις των τύπων του χαρακτήρα και του ακεραίου, επιτρέπεται η αρχικοποίηση ταυτόχρονα με τη δήλωση μίας μεταβλητής κινητής υποδιαστολής.

Η εισαγωγή τιμών στις μεταβλητές κινητής υποδιαστολής γίνεται με χρήση της συνάρτησης **`scanf()`** και του προσδιοριστή **`%f`** (από τη φράση *floating point*). Η πρόταση

`scanf( "%f", &num );`

διαβάζει από το πληκτρολόγιο έναν πραγματικό αριθμό και τον αποδίδει στη μεταβλητή **`num`**. Εάν πρόκειται να αναγνωσθεί αριθμός διπλής ακριβείας, απαιτείται η χρήση του προσδιοριστή **`%lf`**, ενώ στην ανάγνωση αριθμού εκτεταμένης ακριβείας ο προσδιοριστής **`%f`** αντικαθίσταται από τον προσδιοριστή **`%Lf`**.

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 τον 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

απεικονιστούν 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, που ονομάζεται **mantissa**. 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, $d_1 = 1$. 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 ακριβείας 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 ακεραίους και οι υπόλοιπες 129 σε θετικούς, με την ακόλουθη αντιστοιχία δυαδικής και δεκαδικής τιμής:

$$e_{\min} = (00000000)_2 \quad \rightarrow \quad e_{\min} = (-127)_{10}$$
$$e_{\max} = (11111111)_2 \quad \rightarrow \quad e_{\max} = (128)_{10}$$

Με βάση τα παραπάνω, η μέγιστη απόλυτη τιμή πραγματικών αριθμών που μπορεί να επιτευχθεί με μεταβλητή τύπου **`float`** των 32 bits είναι:

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

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

<u>Παρατηρήσεις:</u>

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 τον 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 );
```

οδηγούν στην εμφάνιση στην οθόνη των ακόλουθων αποτελεσμάτων:

```
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** στρογγυλοποιείται στο **40** και παρελείπεται 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 τον μεταγλωττιστή ως **`double`** και δεσμεύουν τον αντίστοιχο χώρο.

2.2.4.1. Παράδειγμα

Να καταστρωθεί πρόγραμμα που να εξετάζει τα μήκη και τα πεδία τιμών των τύπων κινητής υποδιαστολής μονής και διπλής ακρίβειας.

```
#include <stdio.h>
#include <float.h>       /* για τα όρια του float */
int main()
{
```

```
    printf( "The size of type float is %d bytes\n",sizeof(float) );
    printf( " The size of type double is %d bytes\n",sizeof(double) );
    printf( "\nmin float: %e,   max float: %e\n",FLT_MIN,FLT_MAX);
    printf( "\nmin double: %e,   max double: %e\n",DBL_MIN,DBL_MAX);
    printf( "\n\nPress any key to continue" );

    return 0;
}
```

```
The size of type float is 4 bytes
The size of type double is 8 bytes

min float: 1.175494e-038,   max float: 3.402823e+038

min double: 2.225047e-308,   max double: 1.797693e+308
```

Εικόνα 2.3 *Η έξοδος του προγράμματος του παραδείγματος 2.2.4.1*

2.2.5 Ο προσδιοριστής const

Στη γλώσσα C μπορούν να οριστούν και να αρχικοποιηθούν μεταβλητές, για τις οποίες δεν επιτρέπεται η μεταβολή της τιμής τους. Τέτοιες μεταβλητές δηλώνονται, όπως και οι υπόλοιπες, με την προσθήκη του προσδιοριστή **`const`** στα αριστερά του τύπου τους:

```
const double numDouble=46.358767;
const int numInt=46;
const int numChar='G';
```

Στις παραπάνω προτάσεις δηλώνονται και αρχικοποιούνται οι μεταβλητές τριών διαφορετικών τύπων **`numDouble`**, **`numInt`** και **`numChar`**. Οι μεταβλητές αυτές δεν μπορούν να λάβουν άλλη τιμή καθ'όλη τη διάρκεια εκτέλεσης του προγράμματος.

2.3. Είσοδος/Έξοδος προγράμματος

Η είσοδος/έξοδος προγράμματος (input/output) αναφέρεται στις λειτουργίες που γίνονται στις συσκευές εισόδου (συνήθως το πληκτρολόγιο) και εξόδου (συνήθως η οθόνη) του υπολογιστή. Η γλώσσα C δεν διαθέτει εντολή επικοινωνίας με το εξωτερικό περιβάλλον και το έργο αυτό το επιτελούν συναρτήσεις. Προηγουμένως, χρησιμοποιήθηκε το ζεύγος των **`scanf()`** και **`printf()`**, οι οποίες αποτελούν τις μορφοποιούμενες I/O κονσόλας, διότι μπορούν να διαβάζουν και να τυπώνουν δεδομένα σε διάφορες μορφές. Πέραν αυτών, οι συναρτήσεις **`getch()`**, **`getchar()`**, **`putchar()`** διαβάζουν και τυπώνουν με απλούστερο τρόπο δεδομένα τύπου χαρακτήρα, χωρίς όμως να διαθέτουν δυνατότητες μορφοποίησης.

2.3.1. Η συνάρτηση printf()

Στη γλώσσα C η έξοδος των δεδομένων προς το εξωτερικό περιβάλον (συσκευές εξόδου) γίνεται μέσω ροών (αρχείων) εξόδου. Ως *προκαθορισμένη ή πρότυπη ροή εξόδου* (**st**andar**d** **out**put stream – stdout) έχει τεθεί η οθόνη. Η συνάρτηση **`printf()`** (**print f**ormatted data) χρησιμοποιείται για την εμφάνιση μορφοποιούμενης πληροφορίας στην οθόνη. Τυπικά η **`printf()`** εμφανίζει στην οθόνη το περιεχόμενο του ορίσματός της, που περικλείεται μέσα στις παρενθέσεις που την ακολουθούν. Το πρωτότυπο της συνάρτησης έχει την ακόλουθη μορφή:

```
int printf(const char *content, argument_1, argument_2, ..., argument_n);
```

Η συνάρτηση **printf()** εκτυπώνει κατά βάση μία συμβολοσειρά, η οποία εναλλακτικά καλείται αλφαριθμητικό (στο ανωτέρω πρωτότυπο της **printf()** το αλφαριθμητικό το διαχειρίζεται ο δείκτης **content** – περισσότερες πληροφορίες για τους δείκτες στο Κεφάλαιο 6). Ωστόσο, η ισχύς της **printf()** – όπως διαπιστώθηκε στις προηγούμενες παραγράφους– έγκειται στη δυνατότητα που προσφέρει να αντικατασταθούν τμήματα του αλφαριθμητικού από δεδομένα μέσω των *προσδιοριστών* ή *προσδιοριστικών μετατροπής* (conversion specifiers) **%d**, **%c**, **%f** κ.λπ. Οι τιμές των δεδομένων, τα οποία αντιστοιχούν στα ορίσματα **argument_1**, ..., **argument_n**, μπορούν να απεικονιστούν με συγκεκριμένο αριθμό ψηφίων. Επιπρόσθετα, παρέχεται η δυνατότητα να εκτελεστούν περαιτέρω λειτουργίες μορφοποίησης, όπως η αλλαγή γραμμής ή η μετακίνηση κατά μία θέση στηλοθέτη.

Σε περίπτωση επιτυχημένης εκτέλεσης, η συνάρτηση **printf()** επιστρέφει τον αριθμό των χαρακτήρων που εκτυπώνει, αλλιώς επιστρέφει αρνητική τιμή. Στην παρακάτω γραμμή κώδικα

```
x=printf( "This is test no %6d",213 );
```

η ακέραια μεταβλητή **x** δέχεται τον αριθμό των χαρακτήρων που εκτύπωσε η **printf()**, δηλαδή τον αριθμό **22**, καθόσον το αλφαριθμητικό **"This is test no "** περιέχει 17 χαρακτήρες και ο προσδιοριστής **%6d** απαίτησε να εκτυπωθεί ο ακέραιος **213** με 6 χαρακτήρες (3 κενά και ακολούθως οι χαρακτήρες **2**, **1** και **3**).

Για να χρησιμοποιηθεί η συνάρτηση **printf()** απαιτείται η ενσωμάτωση στο πρόγραμμα– μέσω της εντολής προεπεξεργαστή **include**– του αρχείου κεφαλίδας **stdio.h**, καθώς σε αυτό το αρχείο βρίσκεται το πρότυπό της.

2.3.1.1. Μη εκτυπούμενοι χαρακτήρες και σημαίες

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 2.3**:

Χαρακτήρας	Ακολουθία	Χαρακτήρας	Ακολουθία
συναγερμός (κουδούνι)	**\a**	πλάγια γραμμή	****
οπισθοχώρηση μία θέση	**\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**	μηδενικός χαρακτήρας(με ASCII κωδικό 0)	**\0**

Πίνακας 2.3 *Μη εκτυπούμενοι χαρακτήρες και αντίστοιχες ακολουθίες διαφυγής*

Ένα επιπλέον εργαλείο μορφοποίησης των εκτυπούμενων δεδομένων αποτελούν οι *σημαίες* (flags), που παρατίθενται στον **Πίνακα 2.4**. Αριστερά από κάθε σημαία απαιτείται η προσθήκη του συμβόλου **%**.

Σημαία	Λειτουργία
-	Αριστερή στοίχιση της εξόδου στο πεδίο πλάτους (η προκαθορισμένη στοίχιση είναι στα δεξιά)
+	Προσθήκη του προσήμου στις θετικές τιμές

#o	Προσθήκη του **0** μπροστά από οκταδικούς
#x	Προσθήκη του **0x** μπροστά από δεκαεξαδικούς αριθμούς
#X	Προσθήκη του **0X** μπροστά από δεκαεξαδικούς αριθμούς
0	Προσθήκη των απαιτούμενων μηδενικών μπροστά από την τιμή, ώστε να καλυφθεί το πεδίο πλάτους.
κενός χαρακτήρας	Προσθήκη του κενού χαρακτήρα μπροστά από τις μηδενικές τιμές

Πίνακας 2.4 *Σημαίες*

2.3.1.2. Παράδειγμα

Στο πρόγραμμα που ακολουθεί χρησιμοποιούνται διάφορες σημαίες:

```
#include <stdio.h>

int main()
{
   int x=323;

   printf( "1:\t%-5d\n",x );
   printf( "2:\t%+5d\n",x );
   printf( "3:\t% d\n",x );
   printf( "4:\t%#o\n",x );
   printf( "5:\t%#x\n",x );
   printf( "6:\t%#X\n",x );
   printf( "7:\t%05d\n",x );

   return 0;
}
```

```
1:      323
2:      +323
3:      323
4:      0503
5:      0x143
6:      0X143
7:      00323
```

Εικόνα 2.4 *Η έξοδος του προγράμματος του παραδείγματος 2.3.1.2*

Κάθε `printf()` αρχικά εμφανίζει στην οθόνη τους ακέραιους αριθμούς **1**,**2**,...,**7**, ακολουθούμενους από τον χαρακτήρα `':'`. Στη συνέχεια, εμφανίζεται το περιεχόμενο της ακέραιας μεταβλητής `x`, όπως αυτό μορφορποιείται σύμφωνα με την εκάστοτε σημαία. Συγκεκριμένα:

- Η πρώτη `printf()` εμφανίζει την τιμή **323** στοιχισμένη στα αριστερά.
- Στη δεύτερη `printf()` το πλάτος πεδίου είναι 5 και γίνεται χρήση της σημαίας `+`. Εφόσον το **323** καταλαμβάνει 3 θέσεις, στην οθόνη εμφανίζεται ένας κενός χαρακτήρας και ακολούθως το **+323**.
- Στην τρίτη `printf()` η χρήση της σημαίας κενού χαρακτήρα οδηγεί στην εμφάνιση ενός κενού αριστερά της τιμής **323**.
- Στην τέταρτη `printf()` το **323** εμφανίζεται σε οκταδική μορφή.
- Στην πέμπτη και έκτη `printf()` το **323** εμφανίζεται σε δεκαεξαδική μορφή.
- Στην τελευταία `printf()` προστίθενται δύο μηδενικά πριν το **323**.

2.3.2. Η συνάρτηση scanf()

Αντίστοιχα με την έξοδο δεδομένων, και η είσοδος πληροφορίας από το εξωτερικό περιβάλον (συσκευές εισόδου) γίνεται μέσω ροών εισόδου. Ως *προκαθορισμένη* ή *πρότυπη ροή εισόδου* (**st**andar**d** **in**put stream – stdin) έχει τεθεί το πληκτρολόγιο. Η συνάρτηση **`scanf()`** (**scan** **f**ormatted data) χρησιμοποιείται για την ανάγνωση δεδομένων από το πληκτρολόγιο και την αποθήκευσή τους σε μεταβλητές, σύμφωνα με τους προσδιοριστές που αντιστοιχούν στα ορίσματα της συνάρτησης. Το πρωτότυπο της συνάρτησης έχει την ακόλουθη μορφή:

```
int scanf(const char *content, argument_1, argument_2, ..., argument_n);
```

Θα πρέπει να προσεχθεί ότι τα ορίσματα **`argument_1`**, ..., **`argument_n`**, δεν είναι μεταβλητές αλλά οι διευθύνσεις μνήμης των μεταβλητών, στις οποίες θα αποθηκευτούν τα δεδομένα. Το αλφαριθμητικό μορφοποίησης που διαχειρίζεται ο **`content`** περιέχει τόσα προσδιοριστικά όσα είναι και τα προς ανάγνωση δεδομένα. Αντίστοιχος είναι και ο αριθμός των ορισμάτων **`argument_1`**, ..., **`argument_n`**. Κατά συνέπεια, η ακόλουθη γραμμή κώδικα

```
scanf( "%d %d",&x,&y );
```

θα έχει ως αποτέλεσμα να αναγνωσθούν δύο ακέραιοι και να αποθηκευτούν στις θέσεις μνήμες που καταλαμβάνουν οι ακέραιες μεταβλητές **`x`** και **`y`**. Για να γίνει η εκχώρηση τιμής στη μεταβλητή, επιβάλλεται η εισαγωγή του τελεστή διεύθυνσης **`&`** πριν το όνομα της μεταβλητής (με εξαίρεση τα ονόματα αλφαριθμητικών, στα οποία θα αναφερθούμε στο Κεφάλαιο 5).

Σε περίπτωση επιτυχημένης εκτέλεσης, η συνάρτηση **`scanf()`** επιστρέφει τον αριθμό των αναγνωσθέντων δεδομένων. Εάν κάποιο δεδομένο δεν μπορεί να αναγνωσθεί, η **`scanf()`** τερματίζεται και οι μη αναγνωσθείσες τιμές παραμένουν στο **stdin**. Το πρωτότυπο της συνάρτησης βρίσκεται στο αρχείο κεφαλίδας **`stdio.h`**.

Η συνάρτηση **`scanf()`** απαιτεί προσοχή στη χρήση της, καθώς παρουσιάζει ορισμένα ιδιαίτερα χαρακτηριστικά:

- Η κλήση της συνάρτησης **`scanf()`** διακόπτει τη ροή εκτέλεσης του προγράμματος, έως ότου ο χρήστης πληκτρολογήσει τα δεδομένα και πατήσει το πλήκτρο επαναφοράς (**ENTER**).
- Εάν στο τέλος του αλφαριθμητικού μορφοποίησης προστεθεί η ακολουθία διαφυγής αλλαγής γραμμής (**`'\n'`**), τότε η συνάρτηση **`scanf()`** υποχρεώνεται να προχωρήσει στην ανάγνωση του επόμενου μη κενού χαρακτήρα. Κατά συνέπεια, η ακόλουθη γραμμή κώδικα

```
scanf( "%d\n",&x );
```

οδηγεί στην ανάγνωση ενός ακεραίου και στην παύση της εκτέλεσης του προγράμματος, καθώς αναμένεται η εισαγωγή από τον χρήστη ενός μη κενού χαρακτήρα.

- Όταν η συνάρτηση **`scanf()`** χρησιμοποιείται για την ανάγνωση αριθμητικών τιμών, οι «λευκοί χαρακτήρες» που πιθανόν υπάρχουν πριν την αριθμητική τιμή (οι χαρακτήρας του κενού, της αλλαγής γραμμής και του στηλοθέτη), αγνοούνται.
- Επιτρέπεται η χρήση προσδιοριστή πλάτους μέσα στο αλφαριθμητικό μορφοποίησης. Κατά συνέπεια, στην ακόλουθη γραμμή κώδικα

```
scanf( "%4d",&x );
```

εάν ο χρήστης πληκτρολογήσει **57937425**, θα αποθηκευτούν στην ακέραια μεταβλητή **`x`** οι τιμές των πρώτων τεσσάρων ψηφίων, δηλαδή το περιεχόμενο της **`x`** θα γίνει **5793**.

- Η συνάρτηση **`scanf()`** μπορεί να λάβει προδιαγραφές μετατροπής, δηλαδή να θέσει περιορισμούς στους χαρακτήρες που μπορεί να αναγνώσει, μέσω του προσδιοριστή **`[]`**. Επιπλέον, με χρήση του συμβόλου **`-`** μπορούν να προσδιοριστούν ομάδες αποδεκτών χαρακτήρων, ενώ με χρήση του συμβόλου **`^`** μπορούν να αποκλειστούν χαρακτήρες. Για παράδειγμα, ο προσδιοριστής μορφοποίησης **`%[K-W]`** επιτρέπει την

καταχώρηση μόνο των χαρακτήρων **'K'**,**'L'**,...,**'W'**, ενώ ο προσδιοριστής **%[^K-W]** δεν επιτρέπει την ανάγνωση των χαρακτήρων αυτών.

- Εάν μπροστά από έναν προσδιοριστή τοποθετηθεί αστερίσκος *****, τότε το δεδομένο που θα δοθεί από το πληκτρολόγιο θα αναγνωσθεί μεν, ωστόσο δεν θα αποθηκευτεί στο αντίστοιχο όρισμα της συνάρτησης.

- Η συνάρτηση **scanf()** μπορεί να εμφανίσει προβληματική συμπεριφορά, όταν επιχειρηθεί η ανάγνωση δεδομένων τύπου χαρακτήρα με διαδοχικές κλήσεις της. Αυτό οφείλεται στο ότι κατά την πληκτρολόγηση του **ENTER**, ο χαρακτήρας αλλαγής γραμμής αποθηκεύεται στο **stdin** και η επόμενη κλήση της **scanf()** τον ανασύρει και τον αποθηκεύει στο όρισμα εκείνο, το οποίο ανέμενε να αποθηκεύσει τον χαρακτήρα που θα πληκτρολογούσε ο χρήστης.

2.3.2.1. Παράδειγμα

Στο πρόγραμμα που ακολουθεί αποτυπώνεται η χρήση της συνάρτησης **scanf()** με διάφορους τρόπους:

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

int main()
{
   int v1,v2;
   char c1,c2;
   /* 1η ομάδα προτάσεων */
   printf( "\n\tGive two integers: " );
   scanf(" \t%d,%d",&v1,&v2 );
   printf(" \n\tv1=%d, v2=%d",v1,v2 );
   /* 2η ομάδα προτάσεων */
   printf( "\n\n\tGive a character: " );
   scanf( "\t%c",&c1 );
   printf( "\n\tc1=%c",c1 );
   /* 3η ομάδα προτάσεων */
   printf( "\n\n\tGive another integer: " );
   scanf( "\t%d",&v1 );
   printf( "\n\tv1=%d",v1 );
   /* 4η ομάδα προτάσεων */
   printf( "\n\n\tGive two new characters: " );
   scanf( "%c",&c1 );
   scanf( "%c",&c2 );
   printf( "\n\tc1=%c\tc2=%c",c1,c2 );

   return 0;
}
```

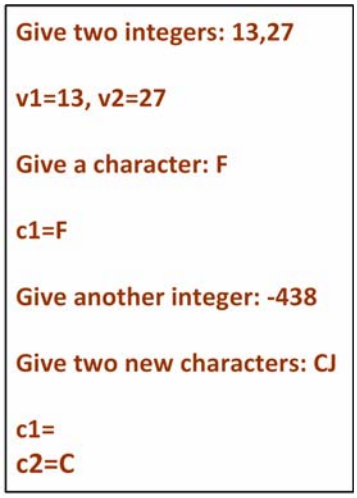
- Στην πρώτη ομάδα προτάσεων η συνάρτηση **scanf()** διαβάζει δύο ακεραίους, τις τιμές των οποίων αποδίδει στις ακέραιες μεταβλητές **v1** και **v2**.

- Στη δεύτερη ομάδα προτάσεων αναγιγνώσκεται ένας χαρακτήρας και αποθηκεύεται στη μεταβλητή τύπου χαρακτήρα **c1**.

- Στην τρίτη ομάδα προτάσεων γίνεται η ανάγνωση και αποθήκευση ενός ακεραίου.

- Στην τέταρτη ομάδα προτάσεων αναδεικνύεται το πρόβλημα στη λειτουργία της **scanf()** που διατυπώθηκε προηγουμένως: ενώ επιχειρείται να αναγνωσθούν οι χαρακτήρες **'C'**,...,**'J'**, στη μεταβλητή **c1** αποθηκεύεται ένας κενός χαρακτήρας και στη μεταβλητή **c2** αποθηκεύεται το **'C'**. Το παράδοξο αποτέλεσμα εξηγείται από το γεγονός ότι στο τέλος της τρίτης σειράς προτάσεων, η πληκτρολόγηση του **ENTER** μετά την τιμή **-438** οδήγησε την αποθήκευση του χαρακτήρα αλλαγής γραμμής στο stdin. Όταν,

ακολούθως, πληκτρολογήθηκαν οι χαρακτήρες `'C'`,...,`'J'`, αυτοί τοποθετήθηκαν με τη σειρά στο **stdin**. Έτσι, όταν έγιναν οι εκχωρήσεις τιμής, πρώτα αποδόθηκε στη μεταβλητή `c1` ο αποθηκευθείς λευκός χαρακτήρας της αλλαγής γραμμής, μετά αποθηκεύτηκε στη μεταβλητή `c2` ο χαρακτήρας `'C'`, ενώ ο χαρακτήρας `'J'` παρέμεινε στο **stdin**.



Εικόνα 2.5 *Η έξοδος του προγράμματος του παραδείγματος 2.3.2.1*

2.3.3. Η συνάρτηση getch()

Η συνάρτηση `getch()` διαβάζει έναν χαρακτήρα από την προκαθορισμένη είσοδο. Αναμένει, έως ότου πατηθεί ένα πλήκτρο, και στη συνέχεια επιστρέφει την τιμή του, χωρίς όμως να εμφανίζεται στην οθόνη το πλήκτρο που πατήθηκε. Το πρωτότυπο της συνάρτησης `getch()` βρίσκεται στο αρχείο κεφαλίδας `stdio.h` και η δήλωσή της είναι

```
int getche(void);
```

Η συνάρτηση `getch()` επιστρέφει μεν έναν ακέραιο, αλλά μόνο στο πρώτο byte (byte χαμηλής τάξης) περιέχεται ο χαρακτήρας. Η χρήση ακεραίων γίνεται για λόγους συμβατότητας με τον αρχικό μεταγλωττιστή της UNIX C.

2.3.4. Οι συναρτήσεις getchar() – putchar()

Οι συναρτήσεις `getchar()` (**get char**acter) και `putchar()` (**put char**acter) είναι το αρχικό ζεύγος εισόδου– εξόδου χαρακτήρων και βασίζονται στο UNIX. Τα πρωτότυπά τους βρίσκονται στο αρχείο κεφαλίδας `stdio.h` και οι δηλώσεις τους είναι

```
int getchar(void);
int putchar(int c);
```

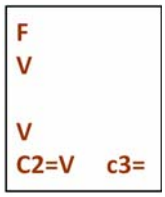
Η συνάρτηση `getchar()` διαβάζει έναν χαρακτήρα από την προκαθορισμένη είσοδο και τον επιστρέφει στο πρόγραμμα. Αποτελεί παραλλαγή της `getch()`. Ένα μειονέκτημα της συνάρτησης αυτής είναι ότι κρατά την είσοδο στην περιοχή προσωρινής αποθήκευσης, μέχρι να πατηθεί **ENTER**. Έτσι, μετά την επιστροφή της `getchar()` περιμένουν ένας ή περισσότεροι χαρακτήρες στην ουρά εισόδου του **stdin**.

Η συνάρτηση `putchar()` εμφανίζει στην οθόνη τον χαρακτήρα που έχει ως όρισμα, στην τρέχουσα θέση του δρομέα. Επιστρέφει μεν έναν ακέραιο, αλλά το byte χαμηλής τάξης είναι αυτό που περιέχει τον χαρακτήρα. Όπως συνέβη και με τις προηγούμενες συναρτήσεις, η χρήση ακεραίων γίνεται για λόγους συμβατότητας με τον αρχικό μεταγλωττιστή της UNIX C.

2.3.4.1. Παράδειγμα

Στο πρόγραμμα που ακολουθεί αποτυπώνεται η χρήση των συναρτήσεων **getch()**, **getchar()** και **putchar()**:

```
#include <stdio.h>

int main()
{
   char c1,c2,c3;

   c1=getch();
   putchar(c1);
   printf( "\n" );
   c2=getchar();
   printf( "\n" );
   putchar(c2);
   c3=getchar();
   printf( "\nc2=%c\tc3=%c",c2,c3 );

   return 0;
}
```



Εικόνα 2.6 *Η έξοδος του προγράμματος του παραδείγματος 2.3.4.1*

Η συνάρτηση **getch()** διαβάζει τον χαρακτήρα **'F'** και τον αποδίδει στη μεταβλητή **c1**, αλλά δεν τον εμφανίζει. Η εμφάνισή του οφείλεται στη χρήση της **putchar()** με όρισμα τη μεταβλητή **c1**. Ακολούθως, η **getchar()** διαβάζει τον χαρακτήρα **'V'** και τον εμφανίζει στην οθόνη. Η δεύτερη εμφάνιση του **'V'** είναι αποτέλεσμα της **putchar(c2)**.

Η πρόταση **c3=getchar()** θα έπρεπε να έχει ως αποτέλεσμα την εκ νέου ανάγνωση χαρακτήρα, ωστόσο τα αποτελέσματα δεν είναι τα αναμενόμενα: από την προηγούμενη χρήση της **getchar()** παρέμενε αποθηκευμένος στον χώρο προσωρινής αποθήκευσης ο λευκός χαρακτήρας αλλαγής γραμμής, που αντιστοιχεί στο πλήκτρο **ENTER**. Αυτός ο χαρακτήρας αποδόθηκε στη μεταβλητή **c3**, γεγονός που αναδεικνύει το ζήτημα που ανακύπτει σε διαδοχικές εκχωρήσεις τιμών σε μεταβλητές με χρήση της **getchar()**.

Ένας τρόπος να ξεπεραστεί το πρόβλημα, είναι να παρεμβληθεί η εντολή **fflush(stdin)** ανάμεσα στις προτάσεις **c2=getchar()** και **c3=getchar()**. Η **fflush()** καθαρίζει τον χώρο προσωρινής αποθήκευσης. Ωστόσο, επειδή σύμφωνα με το πρότυπο της γλώσσας C η συμπεριφορά της **fflush()** είναι απροσδιόριστη, μία άλλη λύση είναι να παρεμβληθεί μία **getchar()** χωρίς να εκχωρεί το αναγνωσθέν δεδομένο σε κάποια μεταβλητή, ώστε να διαβάσει τον αποθηκευμένο χαρακτήρα αλλαγής γραμμής:

```
#include <stdio.h>

int main()
{
```

```
    char c1,c2,c3;

    c1=getch();
    putchar(c1);
    printf( "\n" );
    c2=getchar();
    printf( "\n" );
    putchar(c2);
    getchar();
    c3=getchar();
    printf( "\nc2=%c\tc3=%c",c2,c3 );

    return 0;
}
```

```
F
V

VG
C2=V    c3=G
```

Εικόνα 2.7 *Η νέα έξοδος του προγράμματος του παραδείγματος 2.3.4.1*

2.3.5. Η συνάρτηση kbhit()

Η συνάρτηση **kbhit()** (**k**ey**b**oard **hit**) ελέγχει κατά πόσο ο χρήστης έχει πατήσει κάποιο πλήκτρο. Εφόσον έχει πατήσει κάποιο πλήκτρο, η συνάρτηση επιστρέφει ως αληθής (δηλαδή επιστρέφεται μη μηδενικός ακέραιος), σε αντίθετη περίπτωση επιστρέφει ως ψευδής (δηλαδή επιστρέφεται το μηδέν). Η συνάρτηση **kbhit()** χρησιμοποιείται κυρίως για να διακόπτει ο χρήστης τη ροή του προγράμματος κατά το δοκούν.

Το πρωτότυπο της **kbhit()** βρίσκεται στο αρχείο κεφαλίδας **stdio.h** και η δήλωσή της είναι

```
int kbhit(void);
```

2.3.6. Η συνάρτηση exit()

Η συνάρτηση **exit()** επιτρέπει τον άμεσο τερματισμό ενός προγράμματος. Οι προτάσεις που βρίσκονται κάτω από την **exit()** δε θα εκτελεστούν. Το πρωτότυπο της **exit()** βρίσκεται στο αρχείο κεφαλίδας **stdlib.h** και η δήλωσή της είναι

```
void exit(int status);
```

Η μεταβλητή **status** υποδηλώνει κατά πόσο ο τερματισμός είναι κανονικός ή όχι. Σε περίπτωση κανονικού τερματισμού η **exit()** καλείται ως

```
exit(0);        ή      exit(EXIT_SUCCESS);
```

ενώ στην περίπτωση τερματισμού λόγω σφάλματος καλείται ως

```
exit(EXIT_FAILURE);
```

Θα πρέπει να σημειωθεί ότι το πρότυπο της γλώσσας C συσχετίζει μόνο το **EXIT_FAILURE** με την περίπτωση τερματισμού λόγω σφάλματος, ωστόσο από τους προγραμματιστές χρησιμοποιούνται και μη μηδενικοί ακέραιοι (συνήθως **1** και **-1**) για να υποδηλώσουν τις ενδεχόμενες περιπτώσεις σφάλματος.

2.4. Η έννοια της έκφρασης και του τελεστή στη γλώσσα C

Στον προγραμματισμό οι τελεστές (operators) έχουν την ίδια έννοια που έχουν και στα μαθηματικά, αποτελώντας 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 τον 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 2.5**:

Κατηγορία	Ενδεικτικοί τελεστές
αριθμητικοί	`+ - * /`
λογικοί	`&& || !`
συσχετιστικοί	`> >= = = !=`
διαχείρισης bits	`>> & ! ^`
διαχείρισης μνήμης	`& [ ] . ->`

Πίνακας 2.5 *Κατηγορίες τελεστών*

Τα σύμβολα των συνηθέστερων δυαδικών τελεστών στη C παρατίθενται στον **Πίνακα 2.6**:

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 2.6 *Σύμβολα δυαδικών τελεστών*

2.4.1. Κατηγορίες τελεστών– σημειογραφία

Η γλώσσα C δίνει τη δυνατότητα να επιτυγχάνονται διαφορετικές λειτουργίες στην ίδια έκφραση ανάλογα με τη θέση των τελεστών ανάμεσα στους τελεστέους. Για τον 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`.
- Η σημειογραφία ***παρελκόμενου τελεστή*** (postfix notation), όταν ο τελεστής τοποθετείται μετά από τους τελεστέους, όπως στην έκφραση `x +`.

Παρατηρήσεις:

1. Οι τελεστές είναι προτιμότερο να μην 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 γλώσσα C υπάρχει διάκριση ανάμεσα στη διαίρεση ακεραίων και στη διαίρεση αριθμών κινητής υποδιαστολής. **Στη διαίρεση ακεραίων το αποτέλεσμα προέρχεται από το σχήμα (Διαιρετέος = Πηλίκο*Διαιρέτης + Υπόλοιπο)**, επομένως η διαίρεση του **5** με το **2** παράγει το ακέραιο πηλίκο **2** κι όχι το **2.5**. Για να ληφθεί ως αποτέλεσμα αριθμός κινητής υποδιαστολής, τουλάχιστον ένας από τους τελεστέους πρέπει να είναι αριθμός κινητής υποδιαστολής: η έκφραση `5.0/2` υπολογίζεται ως **2.5**.

2.5. Κατηγορίες εκφράσεων της γλώσσας C– προτεραιότητα και προσεταιριστικότητα

Οι εκφράσεις της γλώσσας 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 (η έννοια του δείκτη θα αναλυθεί στο κεφάλαιο 6), 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+9)>=k) && j)
```

η έκφραση `n+9` είναι φωλιασμένη στην έκφραση `(n+9)>=k`, η οποία με τη σειρά της είναι φωλιασμένη στη συνολική έκφραση. Μία άλλη περίπτωση δυσχέρειας στον υπολογισμό είναι η διαδοχική παράθεση τελεστών: `4*9-12`, η οποία μπορεί να υπολογιστεί είτε ως `(4*9)-12=24` είτε ως `4*(9-12)=-12`, οδηγώντας σε διαφορετικά αποτελέσματα.

Για να αντιμετωπιστούν 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 δρουν 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 `9-3-2` υπολογίζεται ως `(9-3)-2`. Οι τελεστές `+`, `-` ,`*`, `/` είναι όλοι αριστερής προσεταιριστικότητας.

Για τη γλώσσα 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**. 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 2.7**, όπου οι τελεστές έχουν τεθεί με *σειρά φθίνουσας προτεραιότητας*:

Τελεστές	Είδος προσεταιριστικότητας
`()` `[]` `->`	από αριστερά προς τα δεξιά
`!` `~` `++` `--` `+` `-` `*` `&`	από δεξιά προς τα αριστερά

(τύπος) sizeof	
* / % (αριθμητικοί τελεστές)	από αριστερά προς τα δεξιά
+ - (αριθμητικοί τελεστές)	»
<< >>	»
< <= > >=	»
== !=	»
&	»
^	»
|	»
&&	»
||	»
?:	από δεξιά προς τα αριστερά
= += -= *= %= &= ^= |= <<= >>=	»
	από αριστερά προς τα δεξιά

Πίνακας 2.7 *Προτεραιότητα και προσεταιριστικότητα τελεστών*

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

- Για να προσδιοριστεί 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.

2.5.1. Παράδειγμα

Με χρήση του **Πίνακα 2.7** να υπολογιστούν οι εκφράσεις:
(α) `x=17-2*8`
(β) `y=17-2-8`

(α) Βάσει της προτεραιότητας, πρώτα θα εκτελεστεί ο πολλαπλασιασμός `2*8` και το γινόμενο που θα προκύψει θα αποτελέσει τελεστή στην αφαίρεση από το `17`. Το τελικό αποτέλεσμα `1` ανατίθεται στη μεταβλητή `x` που βρίσκεται αριστερά του τελεστή ανάθεσης `=`.

`x=17-(2*8)=17-16=1`

(β) Εφόσον εμφανίζονται δύο στιγμιότυπα του τελεστή `-`, οι κανόνες προσεταιριστικότητας καθορίζουν την εκτέλεση των πράξεων από τα αριστερά προς τα δεξιά:

`y=(17-2)-8=15-8=7`

2.6. Τελεστές μοναδιαίας αύξησης και μείωσης

Ο τελεστής μοναδιαίας αύξησης (increment operator) συμβολίζεται `++`. Με χρήση αυτού του τελεστή η έκφραση `num=num+1` είναι ισοδύναμη με την έκφραση `num++`.

Αντίστοιχα, ο τελεστής μοναδιαίας μείωσης (decrement operator) συμβολίζεται `--` και η έκφραση `num=num-1` είναι ισοδύναμη με την έκφραση `num--`.

2.6.1. Παράδειγμα

Προπορευόμενοι και παρελκόμενοι τελεστές μοναδιαίας αύξησης και μείωσης: να υπολογιστούν οι τιμές των **x** και **y** στις προτάσεις του **Πίνακα 2.8**, οι οποίες εκτελούνται διαδοχικά.

Πρόταση	Τιμή x	Τιμή y
`int x=10, y=15;`	10	15
`++x;`	11	15
`y=--x;`	10	5
`y=x-- + y;`	9	15
`y=y - x--;`	8	6

Πίνακας 2.8

2.6.2. Παράδειγμα

Να προσδιοριστεί η τιμή των **x**, **y** και **z** μετά την εκτέλεση καθεμιάς από τις παρακάτω προτάσεις, θεωρώντας ότι πριν την εκτέλεση της κάθε πρότασης οι τιμές των **x** και **y** είναι το **10** και το **15** αντίστοιχα.

(α) `z=++x + y;`

(β) `z=--x + y;`

(γ) `z=x++ + y;`

(δ) `z=x-- + y;`

(ε) `z=x-- + ++y;`

(στ) `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)*). Στις προτάσεις *(ε)* και *(στ)* εμφανίζονται και προπορευόμενος και παρελκόμενος τελεστής, οπότε το σύστημα χειρίζεται την κάθε περίπτωση ξεχωριστά, σύμφωνα με τους προαναφερθέντες κανόνες. Τα αποτελέσματα παρατίθενται στον **Πίνακα 2.9**:

Πρόταση	Τιμή x	Τιμή y	Τιμή z
`z=++x + y;`	11	15	26
`z=--x + y;`	9	15	24
`z=x++ + y;`	11	15	25
`z=x-- + y;`	9	15	25
`z=x-- + ++y;`	9	16	26
`z=++x + y--;`	11	14	26

Πίνακας 2.9

2.7. Τελεστές ανάθεσης

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

- `x*=100;` Εκτελεί την πράξη του πολλαπλασιασμού μεταξύ των `x` και **100** και εκχωρεί το αποτέλεσμα στον τελεστέο `x`. Αντιστοιχεί στην πρόταση `x=x*100;`
- `x*=y+12;` Αντιστοιχεί στην πρόταση `x=x*(y+12);` κι όχι στην πρόταση `x=x*y+12;`

Τελεστές ανάθεσης δημιουργούν και οι τελεστές διαχείρισης δυαδικών ψηφίων. Οι τελεστές αυτοί είναι: `>>=`, `<<=`, `&=`, `^=`, και `|=` (βλ. §2.10).

Οι τελεστές ανάθεσης σε συνδυασμό με τους τελεστές μοναδιαίας αύξησης/μείωσης γίνονται αιτία δημιουργίας παρενεργειών, για τον 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 `x` σε εκφράσεις, όπως `x= ++x + 4;`

2.8. Συσχετιστικοί τελεστές

Οι συσχετιστικοί τελεστές (relational operators) συγκρίνουν δύο τελεστέους. Οι βασικοί τελεστές της κατηγορίας αυτής παρατίθενται στον **Πίνακα 2.10**:

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

Πίνακας 2.10 *Σύμβολα συσχετιστικών τελεστών*

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

Παρατήρηση: Συγκρίνοντας τους αριθμητικούς με τους συσχετιστικούς τελεστές προκύπτει ότι και οι δύο χρησιμοποιούν αριθμητικούς τελεστέους, π.χ. `(num+10)` και `(num<10)`, όπου `num` είναι μία ακέραια μεταβλητή. Ωστόσο, οι αριθμητικοί τελεστές μπορούν να δώσουν ως αποτέλεσμα οποιονδήποτε αριθμό (για κάθε τιμή του `num` η πρόταση `(num+10)` δίνει μία άλλη τιμή), ενώ οι συσχετιστικοί 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)).

2.9. 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 2.11**, ενώ στον **Πίνακα 2.12** περιγράφεται ο τρόπος λειτουργίας τους (πίνακας αληθείας):

Τελεστής	Δράση
`&&`	λογικό AND

`||`	λογικό OR
`!`	λογικό NOT

Πίνακας 2.11 *Σύμβολα λογικών τελεστών*

x	y	x && y	x || y	!x
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

Πίνακας 2.12 *Πίνακας αληθείας των λογικών τελεστών*

2.9.1. Παράδειγμα

Για `x=16` και `y=-6` να υπολογιστούν οι εκφράσεις:

(α) `(x+9)<(13-y)`

(β) `(x<5) || (y>10)`

(γ) `(x>=7) && (!((x-15)<y))`

Αντικαθιστώντας τις αριθμητικές τιμές προκύπτει:

(α) `(16+9)<(13-(-6))` → `25<19` → **FALSE**

(β) `(16<5) || (-6>10)` → `FALSE || FALSE` → **FALSE**

(γ) `(16>=7) && (!((16-15)<(-6)))` → `TRUE && (!(1<(-6)))` → `TRUE && (!FALSE)` → `TRUE && TRUE` → **TRUE**

2.9.2. Παράδειγμα

Να εξαχθεί η έξοδος του ακόλουθου προγράμματος:

```
#include <stdio.h>

int main()
{
   int x1=4,x2=-17,x3=10,x4;
   /* 1η ομάδα προτάσεων */
   x3=++x1 - --x2;
   printf("x1=%d, x2=%d, x3=%d\n",x1,x2,x3);
   /* 2η ομάδα προτάσεων */
   x4=((x1<1000) || !((x2-x3)>(x1=x1+8)));
   printf("x1=%d, x4=%d, %d\n",x1,x4,((x1>3) && !(x4/x2)));

   return 0;
}
```

```
x1=5, x2=-18, x3=23
x1=5, x4=1, 1
```

Εικόνα 2.8 *Η έξοδος του προγράμματος του παραδείγματος 2.9.2*

- Στην πρώτη ομάδα προτάσεων, λόγω των προπορευόμενων τελεστών η αρχική τιμή της **x1** αυξάνεται κατά **1** και της **x2** μειώνεται κατά **1**, με αποτέλσμα οι νέες τιμές τους να είναι **5** και **-18**. Οι τιμές αυτές αφαιρούνται μεταξύ τους και το αποτέλεσμα (**5 - (-18) = 23**) εκχωρείται στη μεταβλητή **x3**.
- Στη δεύτερη ομάδα προτάσεων οι έλεγχοι οδηγούν στα ακόλουθα:
  - **x1<1000** → **5<1000** → **TRUE**
  - **(x2-x3)>(x1=x1+8))** → **(-18-23)>(x1=5+8))** → **-41>13** → **FALSE**
  - **!((x2-x3)>(x1=x1+8))** → **!FALSE** → **TRUE**
  - **x4=TRUE || TRUE** → **TRUE**, δηλαδή **1**
- Ο τελευταίος προσδιοριστής στην **printf()** της δεύτερης ομάδας προτάσεων αντιστοιχεί στο αποτέλεσμα του ελέγχου: **((x1>3) && !(x4/x2))** → **((5>3) && !(1/23))** → **TRUE && !0** → **TRUE && !FALSE** → **TRUE && TRUE** → **TRUE**, δηλαδή **1**

2.10. Τελεστές διαχείρισης δυαδικών ψηφίων

Στη γλώσσα C υπάρχουν έξι τελεστές διαχείρισης δυαδικών ψηφίων (bitwise operators), οι οποίοι επιτρέπουν την εκτέλεση πράξεων σε επίπεδο bit. Οι τελεστές αυτοί βρίσκουν εφαρμογή στα πεδία της κωδικοποίησης και της συμπίεσης δεδομένων. Οι τελεστές της κατηγορίας αυτής παρατίθενται στον **Πίνακα 2.13**, ενώ στον **Πίνακα 2.14** περιγράφεται ο τρόπος λειτουργίας τους (πίνακας αληθείας):

Τελεστής	Δράση
&	λογικό AND
|	λογικό OR
~	συμπλήρωμα ως προς 1
^	eXclusive OR
<<	ολίσθηση προς τα αριστερά
>>	ολίσθηση προς τα δεξιά

Πίνακας 2.13 *Σύμβολα τελεστών διαχείρισης δυαδικών ψηφίων*

x	y	x & y	x | y	x ^ y	~x
1	1	1	1	0	0
1	0	0	1	1	
0	1	0	1	1	1
0	0	0	0	0	

Πίνακας 2.14 *Πίνακας αληθείας των τελεστών διαχείρισης δυαδικών ψηφίων*

- Η ολίσθηση προς τα αριστερά (bit left shift) μετατοπίζει όλα τα bits του αριστερού τελεστέου κατά τόσες θέσεις προς τα αριστερά, όση είναι η τιμή του δεξιού τελεστέου. Τα bits που απομένουν άνευ περιεχομένου στο δεξί τμήμα του τελεσταίου, συμπληρώνονται με μηδενικά. Η ολίσθηση αυτή πολλαπλασιάζει την υφιστάμενη τιμή του αριστερού τελεστέου επί 2^n, όπου **n** είναι ο αριθμός των θέσεων ολίσθησης.
- Η ολίσθηση προς τα δεξιά (bit right shift) μετατοπίζει όλα τα bits του αριστερού τελεστέου κατά τόσες θέσεις προς τα δεξιά, όση είναι η τιμή του δεξιού τελεστέου. Τα bits που απομένουν άνευ περιεχομένου στο αριστερό τμήμα του τελεσταίου, συμπληρώνονται με μηδενικά. Η ολίσθηση αυτή διαιρεί την υφιστάμενη τιμή του αριστερού τελεστέου διά 2^n, όπου **n** είναι ο αριθμός των θέσεων ολίσθησης.
- Όταν χρησιμοποιούνται τελεστές ολίσθησης, είναι προτιμότερο να εφαρμόζεται σε μη προσημασμένες μεταβλητές, καθώς η εφαρμογή τους σε αρνητικούς αριθμούς εξαρτάται από τον εκάστοτε μεταγλωττιστή.

2.10.1. Παράδειγμα

Να υπολογιστούν οι εκφράσεις:
(α) `10101010 & 00001111`

(β) `010101010 | 11110000`

(γ) `11001100 ^ 00111100`

(δ) `~11010110`

(ε) `9<<4`

(στ) `36>>2`

(α) `10101010 & 00001111` → `(1&0)(0&0)(1&0)(0&0)(1&1)(0&1)(1&1)(0&1)` → `00001010`

(β) `01010101 | 11110000` → `(0|1)(1|1)(0|1)(1|1)(0|0)(1|0)(0|0)(1|0)` → `11110101`

(γ) `11001100 ^ 00111100` → `(1^0)(1^0)(0^1)(0^1)(1^1)(1^1)(0^0)(0^0)` → `11110000`

(δ) `~11010110` → `00101001`

(ε) Ο δεκαδικός αριθμός `9` αντιστοιχεί στον δυαδικό αριθμό `00001001` → $b_7b_6b_5b_4b_3b_2b_1b_0$. Μετακίνηση κατά 4 θέσεις προς τα αριστερά σημαίνει ότι θα γίνουν οι ακόλουθες αντικαταστάσεις: $b_7=b_3$, $b_6=b_2$, $b_5=b_1$, $b_4=b_0$, $b_3=b_2=b_1=b_0=0$ και ο αριστερός τελεσταίος γίνεται `10010000`, ο οποίος αντιστοιχεί στον δεκαδικό αριθμό `144`.

(στ) Ο δεκαδικός αριθμός `36` αντιστοιχεί στον δυαδικό αριθμό `00100100` → $b_7b_6b_5b_4b_3b_2b_1b_0$. Μετακίνηση κατά 2 θέσεις προς τα δεξιά σημαίνει ότι θα γίνουν οι ακόλουθες αντικαταστάσεις: $b_0=b_2$, $b_1=b_3$, $b_2=b_4$, $b_3=b_5$, $b_4=b_6$, $b_5=b_7$, $b_6=b_7=0$ και ο αριστερός τελεσταίος γίνεται `00001001`, ο οποίος αντιστοιχεί στον δεκαδικό αριθμό `9`.

2.11. 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 **έμμεση** (implicit conversion), είτε άμεσα από τον προγραμματιστή, οπότε καλείται **ρητή** (explicit conversion).

2.11.1. Έμμεσες μετατροπές

Οι έμμεσες 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` δεν 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 τύπος χαμηλότερης ιεραρχίας μετατρέπεται στον υψηλότερης ιεραρχίας χωρίς να υπάρχει απώλεια πληροφορίας.
- Οι τύποι της γλώσσας κατατάσσονται ιεραρχικά 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 τον **float** σε **double**.

2.11.1.1. Παράδειγμα

Να αναλυθεί η λειτουργία του ακόλουθου προγράμματος:

```
#include <stdio.h>
int 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 );

   return 0;
}
```

```
ch=A, i=65, fl=65.00,
ch=B, i=197, fl=329.00
```

Εικόνα 2.8 *Η έξοδος του προγράμματος του παραδείγματος 2.11.1.1*

(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 στον 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**.

2.11.2. Ρητές μετατροπές– τελεστής typecast

Εκτός από τις αυτόματες μετατροπές, η C επιτρέπει ρητές μετατροπές μίας τιμής σε έναν 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 δεδομένων:

```
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 */
```

2.12. Ο τελεστής sizeof

Ο τελεστής **sizeof** είναι μοναδιαίος και δρα είτε σε σε μεταβλητή ή ολόκληρη έκφραση, π.χ. **sizeof(x+y)** είτε σε τύπο δεδομένων, πχ. **sizeof(int)**.

Σε κάθε περίπτωση επιστρέφει τον αριθμό των bytes που η τιμή της 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 δεν δημιουργεί παρενέργειες στη λειτουργία του **sizeof**.

Ερωτήσεις αυτοαξιολόγησης- ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Η ακολουθία διαφυγής **'\n'** δηλώνει:

(α) Εκτύπωση του χαρακτήρα **'\'**

(β) Εκτύπωση του χαρακτήρα **'n'**

(γ) Μετακίνηση του κέρσορα στην επόμενη γραμμή

(δ) Μετακίνηση του κέρσορα προς τα δεξιά κατά μία θέση στηλογνώμονα

(2) Η μορφοποιούμενη συνάρτηση εκτύπωσης στο κανάλι εξόδου (οθόνη) ονομάζεται:

(α) **print**

(β) **write**

(γ) **fprint**

(δ) **printf**

(3) Ποιο από τα ακόλουθα ονόματα μεταβλητών είναι σωστό;

(α) **_temp_in_F**

(β) **total%**

(γ) **$product**

(δ) **3rd**

(4) Να υπολογιστούν οι τελικές τιμές των **x** και **y** στις ακόλουθες εκφράσεις που εκτελούνται διαδοχικά.

```
int x=10, y=20;
++ x;
y=--x;
y=x-- + y;
y=y - x++;
```

(α) **x=9**, **y=9**

(β) **x=9**, **y=11**

(γ) **x=10**, **y=10**

(δ) **x=10**, **y=11**

(5) Ποια είναι τα αποτελέσματα του ακόλουθου προγράμματος;

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

```
{
   char ch;
   int i;
   float fl;
   fl='A';
   i=ch='B'+2;
   printf( "ch=%c, i=%d, fl=%6.3f\n",ch,i,fl );
   ch=ch+1;
   i=(int)(2.5*fl);
   printf( "ch=%c, i=%d\n",ch,i );

   return 0;
}
```

(α) `ch=D, i=68, fl=65.000`
`ch=E, i=162`
(β) `ch=D, i=65, fl=65.00`
`ch=D, i=65`
(γ) `ch=D, i=D, fl=65.000`
`ch=E, i=162`
(δ) `ch=65, i=68, fl=65.00`
`ch=E, i=162`

Ασκήσεις

Άσκηση 1

Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο τα μήκη των πλευρών ενός ορθογώνιου τριγώνου και θα τυπώνει την υποτείνουσά του (για τον υπολογισμό της τετραγωνικής ρίζας του αριθμού `x` να χρησιμοποιηθεί η συνάρτηση `sqrt(x)`, η οποία δηλώνεται στο αρχείο κεφαλίδας `math.h`).

Άσκηση 2

Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο έναν ακέραιο αριθμό `x`, που εκφράζει δευτερόλεπτα, και θα εμφανίζει στην οθόνη τις ώρες, λεπτά και δευτερόλεπτα που αντιστοιχούν στον `x`. Σημειώνεται ότι ο δυαδικός τελεστής `%` υπολογίζει το υπόλοιπο της διαίρεσης ακεραίων.

Άσκηση 3

Να υπολογιστούν οι παρακάτω εκφράσεις τελεστών διαχείρισης δυαδικών ψηφίων:

(α) `01011101 & 01001111`
(β) `01010101 | 10110100`
(γ) `11000011 ^ 01101100`
(δ) `~10111100`
(ε) `8<<3`
(στ) `48>>2`

Άσκηση 4

Να γραφεί πρόγραμμα, το οποίο:
***(α)* Θα αποθηκεύει σε μεταβλητές τα ακόλουθα:**

- **Την ημέρα γεννήσής σας (π.χ. 12)**
- **Τον μήνα γεννήσής σας (π.χ. 7)**
- **Το έτος γέννησής σας (π.χ. 1996)**
- **Το ύψος σας σε μέτρα (π.χ. 1.85)**
- **Το πρώτο γράμμα του ονόματός σας (π.χ. X)**

***(β)* Τα ανωτέρω στοιχεία θα εμφανίζονται στην οθόνη. Η ημερομηνία θα εμφανίζεται στη μορφή ημέρα/μήνας/έτος. Το ύψος θα εμφανίζεται με δύο δεκαδικά ψηφία.**

Άσκηση 5

Να γραφεί πρόγραμμα, το οποίο:

(α) **Θα διαβάζει από το πληκτρολόγιο δύο ακέραιους αριθμούς, τους οποίους θα αποθηκεύει σε μεταβλητές και θα τους εμφανίζει στην οθόνη.**

(β) **Θα υπολογίζει το άθροισμα και το γινόμενο των δύο αριθμών και θα τα εμφανίζει στην οθόνη.**

(γ) **Θα εμφανίζει στην οθόνη τις απόλυτες τιμές όλων των ανωτέρω αριθμών (η συνάρτηση `abs(x)`, η οποία βρίσκεται στο αρχείο κεφαλίδας `math.h`, επιστρέφει την απόλυτη τιμή ενός ακέραιου αριθμού `x`).**

Άσκηση 6

Να γραφεί πρόγραμμα, το οποίο:

(α) **Θα διαβάζει από το πληκτρολόγιο δύο φυσικούς αριθμούς, τους οποίους θα αποθηκεύει σε μεταβλητές και θα τους εμφανίζει στην οθόνη.**

(β) **Θα υπολογίζει την περίμετρο του ορθογώνιου παραλληλογράμμου που δημιουργείται, όταν τεθούν ως μήκη πλευρών οι ανωτέρω ακέραιοι αριθμοί. Η περίμετρος θα εμφανίζεται στην οθόνη.**

Άσκηση 7

Να εξαχθούν τα αποτελέσματα του παρακάτω προγράμματος:

```
int main()
{
  int x,y,z;

  x=0;
  y=10;
  printf( "\t\n\nx=%d y=%d\n",x,y );

  z=(x>y);
  printf( "x > y is %d\n",z );

  z=(x==y);
  printf( "x==y is %d\n",z );

  z=(x!=y);
  printf( "x != y is %d\n",z );

  z=(x && y);
  printf( "x && y is %d\n",z );

  z=!(x && y) || (x || y);
  printf( "!(x && y) || (x || y) is %d\n",z );

  x = 10;
  printf( "\t\n\n\nx=%d y=%d\n",x,y );

  z=(x>y);
  printf( "x > y is %d\n",z );

  z=(x!=y);
  printf( "x != y is %d\n",z );

  z=(x && y);
  printf( "x && y is %d\n",z );
```

```
    float f1, f2;
    f1=5/(float)x;
    f2=5.0/(float)x;
    printf( "5.0/(float)%d = %f\n",x,f2 );
    printf( "x=%d NOT(x)=%d\n",x,!x );

    z=10+x;
    printf( "10+%d = %d\n",x,z );

    z=10+(!x);
    printf( "10+NOT(%d) = %d\n",x,z );

    return 0;
}
```

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

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

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

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Horton, I. (2006), *Beginning C – from Novice to Professional*, 4th ed., Apress.

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

3. Έλεγχος ροής προγράμματος

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια του ελέγχου ροής προγράμματος. Παρουσιάζονται η ακολουθιακή και επιλεκτική εκτέλεση προτάσεων και οι γλωσσικές κατασκευές της υπό συνθήκη διακλάδωσης και των επαναληπτικών προτάσεων. Τα παραπάνω εξειδικεύονται στη γλώσσα C, όπου μελετώνται οι προτάσεις if–else, switch–case, while, for, do–while. Αναλύεται η λειτουργική ισοδυναμία των τριών ειδών προτάσεων επανάληψης και περιγράφονται οι ένθετοι βρόχοι και οι διακοπτόμενοι βρόχοι.

Λέξεις κλειδιά
ακολουθιακή εκτέλεση προτάσεων, επιλεκτική εκτέλεση προτάσεων, προτάσεις επανάληψης, υπό συνθήκη διακλάδωση, υποθετικός τελεστής, πολλαπλή διακλάδωση, βρόχος που οδηγείται από γεγονός και από μετρητή, ένθετοι βρόχοι, διακοπτόμενοι βρόχοι, if–else, case–switch, while, for, do–while, break, continue, goto

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C – μεταβλητές – εκφράσεις – τελεστές

3.1. Έλεγχος ροής

Τα προγράμματα είναι σύνολα προτάσεων, 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 3.1**:

Σχήμα 3.1 *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 στον 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 γλωσσικές 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:

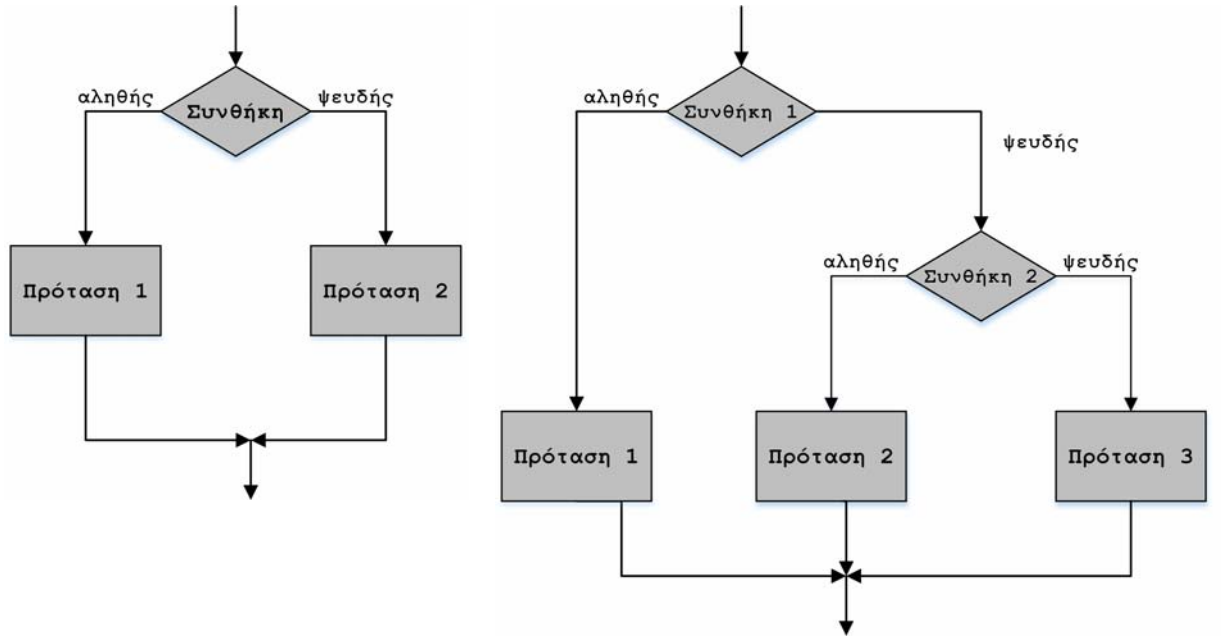
- την ***υπό συνθήκη διακλάδωση*** (conditional branching),
- την ***επανάληψη*** (looping).

3.2. 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 Συνθήκη then Πρόταση 1 else Πρόταση 2
```

Στο **Σχήμα 3.2.α** αναπαρίσταται 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 `Συνθήκη` και, εάν είναι αληθής, επιλέγεται για εκτέλεση η πρόταση `Πρόταση 1`, διαφορετικά η `Πρόταση 2`. Σε κάθε περίπτωση, ο έλεγχος μεταφέρεται στο ένα και μοναδικό σημείο εξόδου στο κάτω μέρος του διαγράμματος.



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

Σχήμα 3.2 *Επιλεκτική εκτέλεση προτάσεων*

Στο **Σχήμα 3.2.β** 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 Συνθήκη 1 then Πρόταση 1
else if Συνθήκη 2 then Πρόταση 2
else Πρόταση 3
```

Οι ανωτέρω εκφράσεις υπολογίζονται σειριακά και η πρώτη που θα δώσει αληθή τιμή οδηγεί στην εκτέλεση της αντίστοιχης πρότασης. Εάν καμία από τις λογικές εκφράσεις δεν δώσει αληθή τιμή, εκτελείται η `Πρόταση 3`.

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 γλώσσας C, στην οποία η επιλογή γίνεται μέσα από ένα σύνολο αμοιβαία αποκλειόμενων επιλογών, όπως θα αναλυθεί σε επόμενη παράγραφο.

3.2.1. Παράδειγμα

Να περιγραφεί με ψευδοκώδικα η διεργασία που πρέπει να ακολουθήσει ο υπολογιστής, για να διαπιστώσει κατά πόσον 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
```

Εναλλακτικά, ξεκινώντας από την περίπτωση το έτος να μην είναι δίσεκτο, η οποία καλύπτει την πλειοψηφία των ετών και συνενώνοντας τους ελέγχους το έτος να διαιρείται με το **100** αλλά όχι με το **400**, έχουμε επιτάχυνση της εκτέλεσης του κώδικα, καθώς – εν γένει – διενεργούνται λιγότεροι έλεγχοι:

```
if ((year%4)!=0) then το έτος δεν είναι δίσεκτο
else if (((year%100)==0) && ((year%400)!==0)) then το έτος δεν είναι
δίσεκτο
else το έτος είναι δίσεκτο
```

3.3. Υπό συνθήκη διακλάδωση if – else

Στη γλώσσα 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) δεν γίνεται καμία ενέργεια.

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

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

μπορεί να αντικατασταθεί 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;
}
```

3.3.1. Παράδειγμα

Να ελεγχθεί κατά πόσον τα ακόλουθα τμήματα κώδικα είναι *λειτουργικά ισοδύναμα*, δηλαδή δεχόμενα ως είσοδο τα ίδια δεδομένα, οδηγούν στα ίδια αποτελέσματα.

```
int george,john;
george=10;
john=george+5;
if (george<20)
{
  george*=2;
}
else
{
  john=1500;
}
printf( "john=%d\n",john );
```

```
int george,john;
george=10;
john=george+5;
if (george<20)
{
  george*=2;
}
if (george>=20)
{
  john = 1500;
}
printf( "john=%d\n",john );
```

Οι πρώτες πέντε γραμμές των δύο τμημάτων κώδικα είναι ίδιες, κατά συνέπεια, όταν ολοκληρωθεί η εκτέλεσή τους οι μεταβλητές **george** και **john** θα έχουν λάβει τις τιμές **20** και **15**, αντίστοιχα. Στην έκτη γραμμή υπάρχει διαφοροποίηση: στο πρώτο τμήμα κώδικα το **else** δεν θα εκτελεστεί, καθώς η συνθήκη του **if** ήταν αληθής, και η μεταβλητή **john** θα διατηρήσει την τιμή της (**15**). Στο δεξί τμήμα κώδικα, όμως, η υπό συνθήκη διακλάδωση **if** είναι καινούρια, η συνθήκη είναι αληθής (**george=2>=20**) και η μεταβλητή **john** θα λάβει νέα τιμή (**1500**). Κατά συνέπεια τα δύο τμήματα κώδικα δεν είναι λειτουργικά ισοδύναμα.

3.3.2. Παράδειγμα

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). Να εξεταστεί μόνο η περίπτωση πραγματικών συντελεστών.

Παρατήρηση: Η απόλυτη τιμή μίας μεταβλητής κινητής υποδιαστολής **x** υπολογίζεται με τη συνάρτηση **fabs(x)**, το πρωτότυπο της οποίας βρίσκεται στο αρχείο κεφαλίας **math.h**. Η τετραγωνική ρίζα υπολογίζεται με τη συνάρτηση **sqrt()**,το πρωτότυπο της οποίας βρίσκεται επίσης στο αρχείο κεφαλίδας **math.h**.

```
#include<stdio.h>
#include<math.h>
int 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 second order equation, a!=0. Try again\n" );
     printf( "\nGive parameter a:"); scanf("%f",&a );
  }  /* τέλος της πρότασης 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 της πρότασης 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 της πρότασης 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 πρότασης else */
}
```

```
This program provides the roots of the second order equation
                        ax^2+bx+c=0

Give parameter a:1

Give parameter b:2

Give parameter c:1
                    1.000*x^2 + 2.000*x + 1.000 = 0
There exists a double root:
r1 = r2 = -1.000
```

Εικόνα 3.1.α *Η έξοδος του προγράμματος του παραδείγματος 3.3.2 για διπλή πραγματική ρίζα*

```
This program provides the roots of the second order equation
                        ax^2+bx+c=0

Give parameter a:1

Give parameter b:1

Give parameter c:2
                    1.000*x^2 + 1.000*x + 2.000 = 0
There exist two conjugate complex roots:
r1 = -0.5000 + j1.323
r1 = -0.5000 - j1.323
```

Εικόνα 3.1β. *Η έξοδος του προγράμματος του παραδείγματος 3.3.2 για συζυγείς μιγαδικές ρίζες*

```
This program provides the roots of the second order equation
                        ax^2+bx+c=0

Give parameter a:1

Give parameter b:1

Give parameter c:-4
                    1.000*x^2 + 1.000*x + -4.000 = 0
There exist two real roots:
r1 = 1.562
r1 = -2.562
```

Εικόνα 3.1.γ *Η έξοδος του προγράμματος του παραδείγματος 3.3.2 για απλές πραγματικές ρίζες*

Στο **Σχήμα 3.1** παρέχονται τρία στιγμιότυπα των αποτελεσμάτων εκτέλεσης του προγράμματος για διαφορετικές τιμές των παραμέτρων της εξίσωσης. Στο **Σχήμα 3.1.α** από την οθόνη ο χρήστης έδωσε τιμές στις παραμέτρους **a**,**b**,**c**, οι οποίες οδήγησαν σε μηδενική διακρίνουσα και σε εκτέλεση της πρότασης διακλάδωσης ***else if (fabs(D)< 1e-10)*** . Κατ' αντιστοιχία, τα αποτελέσματα του **Σχήματος 3.1.β** προήλθαν από εκτέλεση της πρότασης διακλάδωσης ***if(D<0)***, ενώ στο **Σχήμα 3.1.γ** παρουσιάζεται η περίπτωση ύπαρξης πραγματικών ριζών, η οποία σχετίζεται με την εκτέλεση του τελευταίου **else** .

3.3.3. Παράδειγμα

Να γραφεί πρόγραμμα που να υλοποιεί το παράδειγμα 3.2.1.

```
#include <stdio.h>
int 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 );

  return 0;
}
```

```
Give a number: 1588

Year 1588 is a leap year
```

Εικόνα 3.2 *Η έξοδος του προγράμματος του παραδείγματος 3.3.3*

3.4. 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 αναφερθείσες στο Κεφάλαιο 2 σημειογραφίες (προθεματική, ενθεματική και μεταθεματική). 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:

έκφραση 1 ? έκφραση 2 : έκφραση 3

Ουσιαστικά ο υποθετικός τελεστής υλοποιεί μία υποθετική πρόταση. Η τιμή της παραπάνω έκφρασης είναι η τιμή της **έκφρασης 2**, εάν η **έκφραση 1** είναι αληθής, αλλιώς είναι η τιμή της **έκφρασης 3**.

Η **έκφραση 1** αποτελεί τη συνθήκη ελέγχου. Έτσι η έκφραση

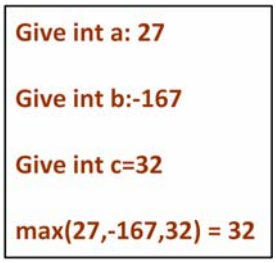
x > z ? x : z

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

3.4.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα διαβάζει από το πληκτρολόγιο τρεις ακέραιους αριθμούς και θα υπολογίζει τον μέγιστο εξ αυτών. Ακολούθως, να τροποποιηθεί ο κορμός του προγράμματος κάνοντας χρήση του υποθετικού τελεστή.

```
#include <stdio.h>
int 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( "\nmax(%d,%d,%d) = %d\n",a,b,c,a );
     else printf( "\nmax(%d,%d,%d) = %d\n",a,b,c,c );
  }
   else if (b>c) printf( "\nmax(%d,%d,%d) = %d\n",a,b,c,b );
   else printf( "\nmax(%d,%d,%d) = %d\n",a,b,c,c );

  return 0;
}
```



Εικόνα 3.3 *Η έξοδος του προγράμματος του παραδείγματος 3.4.1*

Χρησιμοποιώντας τον υποθετικό τελεστή, οι προτάσεις διακλάδωσης και εμφάνισης του αποτελέσματος συνοψίζονται στις ακόλουθες:

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

3.5. Υπό συνθήκη διακλάδωση switch

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

```
switch (έκφραση)
{
  case (σταθερά 1):
    προτάσεις;
```

```
        break;
        case (σταθερά 2):
           προτάσεις;
        break;
        ..............
        case (σταθερά 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 (**σταθερά 1**, **σταθερά 2**, **…**). Ο έλεγχος μεταφέρεται στις προτάσεις που είναι κάτω από τη **σταθερά** 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 της επόμενης **case** ετικέτας. 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** ολοκληρώνεται με τον χαρακτήρα **:** και όχι με ερωτηματικό.
- Κάθε **case** πρέπει να έχει μία σταθερή έκφραση τύπου ακεραίου ή χαρακτήρα.
- 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.

3.5.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο να δίνει τη δυνατότητα στον 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:

```
Διάβασε δύο αριθμούς
Ενημέρωσε τον χρήστη για δυνατές επιλογές
Διάβασε 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
int 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 );

  return 0;
}
```

```
Give first number:12

Give second number: -12.5454

 Select one of the following:
                          1 -> + (addition)
                          2 -> - (subtraction)
                          1 -> * (multiplication)
                          4 -> / (division)
4

        Result: -0.956526
```

Εικόνα 3.4 *Η έξοδος του προγράμματος του παραδείγματος 3.5.1*

3.6. Προτάσεις επανάληψης – βρόχοι

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 μία ομάδα (μπλοκ) εντολών είτε για όσες φορές το επιθυμούμε είτε έως ότου πληρωθεί μία συνθήκη τερματισμού του βρόχου**. Η πλήρωση του κριτηρίου τερματισμού οδηγεί στην περάτωση του βρόχου.

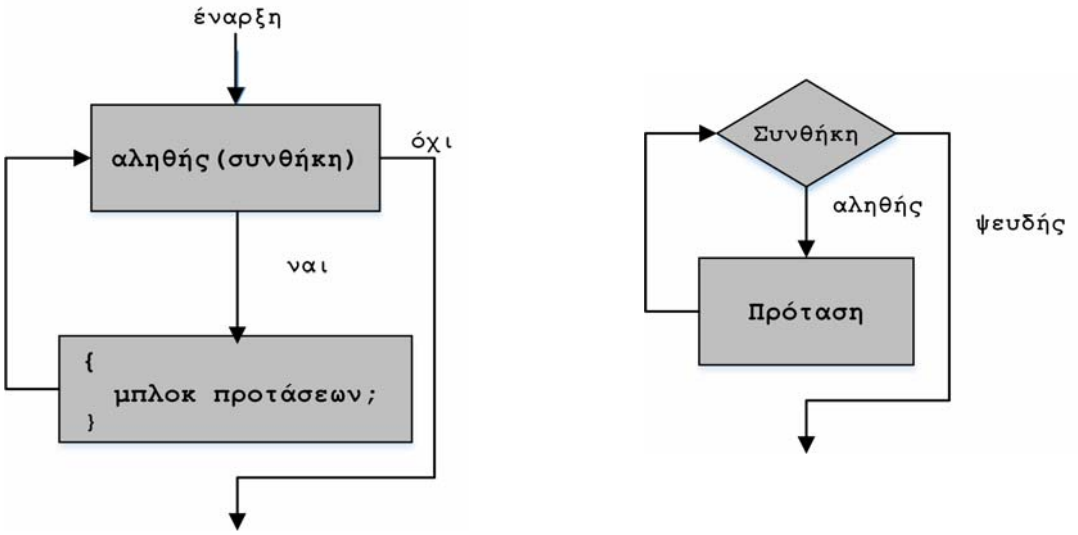
Εάν δεν υπάρχει συγκεκριμένος αριθμός επαναλήψεων ή συνθήκη τερματισμού, ο βρόχος θα εκτελείται αενάως και θα καλείται **ατέρμων βρόχος** (infinite loop), γεγονός που οδηγεί ως επί το πλείστον σε σφάλμα.

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 παρουσιαστούν οι μορφές βρόχων που κυριαρχούν στις γλώσσες προγραμματισμού και στη συνέχεια θα εστιαστούμε στις επαναληπτικές προτάσεις που χρησιμοποιούνται στη γλώσσα C.

3.6.1. 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 3.3.α**, ενώ το διάγραμμα ροής παρατίθεται στο **Σχήμα 3.3.β**. Σημειώνεται ότι το **μπλοκ διάγραμμα** (block diagram) είναι ένα διάγραμμα του κώδικα, στο οποίο τα βασικά τμήματά του ή οι συναρτήσεις του αναπαριστώνται με μπλοκ που συνδέονται με γραμμές. Οι γραμμές υποδεικνύουν τη ροή του ελέγχου προγράμματος, δηλαδή αναπαριστούν τις σχέσεις ανάμεσα στα μπλοκ.



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

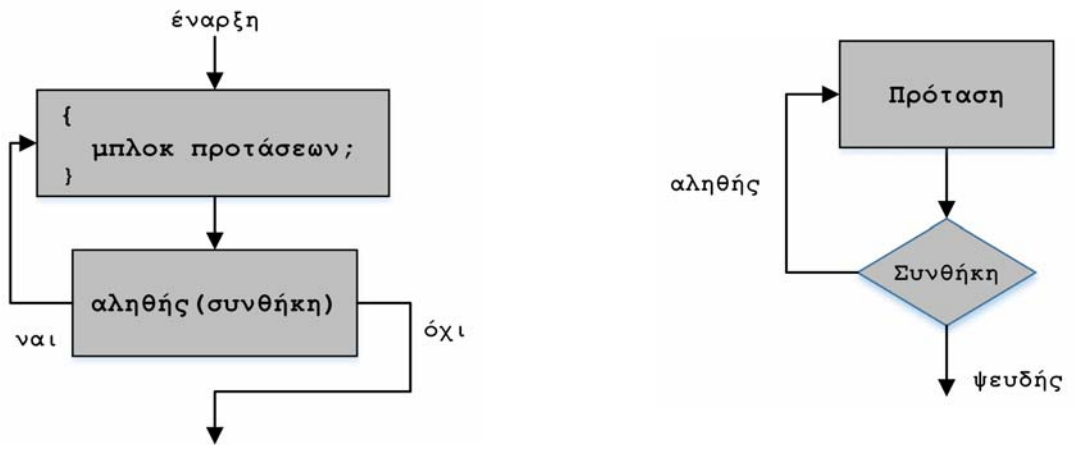
Σχήμα 3.3 *Βρόχος while–do*

Όπως προκύπτει από το **Σχήμα 3.3**, ο βρόχος θα εκτελείται – δηλαδή η **Πρόταση** – για όσες επαναλήψεις η συνθήκη **Συνθήκη** είναι αληθής. Κατά συνέπεια απαιτείται: *α)* πριν την πρώτη επανάληψη η συνθήκη **Συνθήκη** να είναι αληθής και *β)* κατά τη διάρκεια εκτέλεσης του βρόχου να υπάρχει η δυνατότητα, μέσω της **Πρότασης**, να μπορεί να καταστεί ψευδής η **Συνθήκη** για να τερματιστεί ο βρόχος.

3.6.2. 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 3.4.α** και **3.4.β** παρουσιάζονται το μπλοκ διάγραμμα και το διάγραμμα ροής αντίστοιχα.

Από το **Σχήμα 3.4** καθίσταται φανερό ότι ο βρόχος **do-while** διαφέρει από τον 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 **Πρόταση** και, φυσικά, πρέπει να υπάρχει η δυνατότητα, μέσω της **Πρότασης**, να μπορεί να καταστεί η **Συνθήκη** ψευδής για να τερματισθεί ο βρόχος.



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

Σχήμα 3.4 *Βρόχος do–while*

3.7. Βρόχοι με συνθήκη εισόδου στη γλώσσα C

3.7.1. Βρόχος while

Ο βρόχος **while** είναι βρόχος με συνθήκη εισόδου, οδηγούμενος από γεγονός. Η λειτουργία του περιγράφεται εποπτικά από το **Σχήμα 3.3.α** και η σύνταξή του είναι η ακόλουθη:

```
while (συνθήκη)
{
  Προτάσεις μέσα στις οποίες θα μπορεί να αλλάξει 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 τη λειτουργία του βρόχου
```

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 τον βρόχο **do-while** ισχύει η ανάλυση της §3.6.1

Θα πρέπει να σημειωθεί ότι εάν το σώμα του βρόχου αποτελείται από μία πρόταση, δεν απαιτούνται **{}**. Ωστόσο, προτείνεται η χρήση των αγκίστρων σε κάθε περίπτωση, ανεξάρτητα από τον αριθμό των προτάσεων που απαρτίζουν το σώμα του βρόχου.

3.7.1.1. Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα:

```
int counter=10;
int counterLimit=20;
while (counter<counterLimit)
{
  counter++;
  printf( "counter is %d\n",counter );
}
<επόμενη πρόταση>;
```

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

```
counter is 11
```

Ο βρόχος εκτελείται συνολικά **10** φορές (για **counter=10** έως και **19**) και δίνει τα αποτελέσματα:

```
counter is 11
counter is 12
.........
counter is 20
```

Στην ενδέκατη επανάληψη η **counter** έχει λάβει την τιμή **20** και είναι ίση με τη **counterLimit**, οπότε η συνθήκη καθίσταται ψευδής και ο έλεγχος προσπερνά τον βρόχο και προχωρά στην επόμενη πρόταση. Θα πρέπει να σημειωθεί ότι, εάν η συνθήκη ήταν εξαρχής ψευδής (π.χ. **counter=22**), ο βρόχος δεν θα εκτελείτο ούτε μία φορά.

3.7.1.2. Παράδειγμα

Δίνονται οι παρακάτω δύο προτάσεις:
(α) **while (++counter<15) Πρόταση**
(β) **while (counter++<15) Πρόταση**

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

Υπάρχει διαφορά μεταξύ των προτάσεων και αυτή εντοπίζεται στον αριθμό επαναλήψεων. Η *(α)* χρησιμοποιεί την προθεματική σημειογραφία, ενώ η *(β)* τη μεταθεματική. Στην πρόταση *(α)* πρώτα αυξάνεται

η τιμή τής **counter** και η νέα τιμή της συγκρίνεται με το **15**, ενώ στην πρόταση *(β)* πρώτα συγκρίνεται η τιμή της **counter** με το **15** και στη συνέχεια αυξάνεται η τιμή της. Αυτό σημαίνει πως η πρόταση **Πρόταση** θα εκτελεστεί μία φορά παραπάνω στην περίπτωση *(β)*.

3.7.1.3. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:
(α) Θα διαβάζει διαδοχικά χαρακτήρες, έως ότου πληκτρολογηθεί ο χαρακτήρας αλλαγής γραμμής.
(β) Θα μετρά το κενά.
(γ) Μετά το πέρας της ανάγνωσης θα εμφανίζει στην οθόνη τον αριθμό τους.
Ο χαρακτήρας τερματισμού του βρόχου δεν θα προσμετράται στους αναγνωσθέντες χαρακτήρες.

```
#include <stdio.h>
int main()
{
  int numSpaces=0;
  char ch;
  printf( "Give a sentence:\n" );
  ch=getchar();
  while (ch!='\n')
  {
    if (ch==' ') numSpaces++;
    ch=getchar();
  }
  printf( "\n\nThe number of spaces is %d\n",numSpaces);

  return 0;
}
```

```
Give a sentence:

Give_me_the__number___of_spaces

The number of spaces is 8
```

Εικόνα 3.5 *Η έξοδος του προγράμματος του παραδείγματος 3.7.1.3*

Παρατήρηση: Μία συμπαγής γραφή της ανάγνωσης χαρακτήρα και ελέγχου του περιεχομένου του – που χρησιμοποιείται ευρέως στη βιβλιογραφία – είναι η ενσωμάτωση της **getchar()** μέσα στη **while**. Μέσα στη συνθήκη ελέγχου πρώτα εκτελείται η πρόταση στα αριστερά και μετά διεξάγεται ο έλεγχος. Σε αυτήν την περίπτωση προκύπτει ο ακόλουθος λειτουργικά ισοδύναμος κώδικας:

```
#include <stdio.h>
int main()
{
  int numSpaces=0;
  char ch;
  printf( "Give a sentence:\n" );
  while ((ch=getchar())!='\n')
  {
    if (ch==' ') numSpaces++;
  }
  printf( "\n\nThe number of spaces is %d\n",numSpaces);
```

```
    return 0;
}
```

3.7.2. Βρόχος for

Ο βρόχος **for** είναι βρόχος με συνθήκη εισόδου, οδηγούμενος από μετρητή. Η λειτουργία του περιγράφεται εποπτικά από το **Σχήμα 3.3.α** 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 τον μετρητή
Έλεγξε τη συνθήκη
Εάν είναι αληθής
   Εκτέλεσε τις προτάσεις
   Ενημέρωσε τον μετρητή
   Επάνελθε στον έλεγχο της συνθήκης
Αλλιώς ενημέρωσε τον μετρητή και σταμάτησε τη λειτουργία του βρόχου
```

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 );
```

Ο βρόχος **for** στη γλώσσα C παρέχει μεγάλη ευελιξία, καθώς οι εκφράσεις μέσα στις παρενθέσεις μπορούν να έχουν πολλές παραλλαγές:

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

```
for (n=10; n>0; n=n-2) printf( "n=%d\n",n );
```

- Το βήμα καθορίζεται από τον χρήστη:

```
for (n=0; n<40; n=n+15) 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 **'F'** μαζί με τον ASCII κωδικό τους:

```
for (n='A'; n<'F'; 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 );
```

- Στο πρότυπο C99 ο ο μετρητής μπορεί να δηλωθεί μέσα στη δήλωση της **for**. Ωστόσο, απαιτείται ιδαίτερη προσοχή, γιατί μία τέτοια μεταβλητή δεν μπορεί να χρησιμοποιηθεί πουθενά αλλού παρά μόνο μέσα στο μπλοκ προτάσεων της **for**. Δηλαδή, η ακόλουθη επαναληπτική πρόταση είναι σωστή:

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

αλλά η μεταβλητή **count** δεν μπορεί να χρησιμοποιηθεί παρακάτω στο πρόγραμμα.

Θα πρέπει να σημειωθεί ότι, όπως και στην περίπτωση του βρόχου **while**, εάν το σώμα του βρόχου αποτελείται από μία πρόταση, δεν απαιτούνται **{}**, αλλά προτείνεται να χρησιμοποιούνται.

3.7.2.1. Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα:

```
int counter, counterLimit=30;
for (counter=15; counter<counterLimit; counter=counter+4)
{
  printf( "counter is %d\n",counter );
  <προτάσεις>;
}
<επόμενη πρόταση>;
```

Στην πρώτη γραμμή δηλώνονται οι ακέραιες μεταβλητές **counter** και **counterLimit** και αποδίδεται τιμή στη μεταβλητή **counterLimit**. Ακολουθεί η πρόταση επανάληψης **for**, η οποία έχει μετρητή τη μεταβλητή **counter**, αρχική τιμή του μετρητή το **15**, βήμα το **4** και θα εκτελείται, καθόσον ο μετρητής έχει τιμή μικρότερη της **counterLimit**.

```
counter is 15
counter is 19
counter is 23
counter is 27
```

Ο βρόχος εκτελείται συνολικά **4** φορές, για τιμές του μετρητή **15**,**19**,**23**,27. Μετά το πέρας της τέταρτης επανάληψης ο μετρητής έχει λάβει την τιμή **31** και στον έλεγχο της συνθήκης στην πέμπτη επανάληψη η τελευταία είναι ψευδής, οπότε ο έλεγχος προσπερνά τον βρόχο και προχωρά στην επόμενη πρόταση. Η τιμή του μετρητή θα έχει γίνει **31**. Αυτή είναι η τιμή με την οποία ο **counter** συμμετέχει στη συνέχεια του προγράμματος.

3.7.2.2. Παράδειγμα

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 τον βρόχο:

```
#include <stdio.h>

int main()
{
    int x,y;
    for (x=0,y=0; x+y<100; x=x+20,y=y+10)
     printf( "x=%d  y=%d  x+y=%d\n",x,y,x+y );
    printf( "\nOut of loop: x=%d  y=%d\n",x,y );

     return 0;
}
<επόμενη πρόταση>;
```

```
x=0  y=0  x+y=0
x=20  y=10  x+y=30
x=40  y=20  x+y=60
x=60  y=30  x+y=90

Out of loop:  x=80  y=40
```

Εικόνα 3.6 *Η έξοδος του προγράμματος του παραδείγματος 3.7.2.2.*

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.

3.7.3. Ο τελεστής κόμμα (,)

Στο παράδειγμα 3.7.2.2. οι μεταβλητές για τον 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 γι' αυτόν τον 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=1,j=10; i<=9; i++,j=j+10)    printf( "%d\n",i+j );
```

εμφανίζει στην οθόνη τους αριθμούς **11**,**22**,...,**99**, καθώς σε κάθε επανάληψη το **i** αυξάνει κατά **1** και το **j** κατά **10**.

Θα πρέπει να αποφεύγονται προτάσεις όπως η ακόλουθη:

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

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

3.7.4. Μετασχηματισμός βρόχων while–for

Ένας βρόχος **for** μπορεί να μετασχηματιστεί 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;
}
```

3.7.4.1. Παράδειγμα

Τι εμφανίζεται στην οθόνη του υπολογιστή από την εκτέλεση του ακόλουθου βρόχου;

```
for ( n=15;  n>8;  n-- )  printf( "%d,",10%n );
```

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

Ο παραπάνω κώδικας εκτελεί έναν βρόχο **for** με τον μετρητή να φθίνει με μοναδιαίο βήμα, σε κάθε επανάληψη του οποίου εμφανίζεται στην οθόνη το υπόλοιπο της διαίρεσης του **10** με το **n**. Ο βρόχος θα εκτελεστεί για **7** επαναλήψεις και θα εμφανιστούν τα ακόλουθα αποτελέσματα στην οθόνη:

10,10,10,10,10,0,1

Με βάση τη φόρμα μετασχηματισμού της §3.7.4, ο κώδικας με χρήση του βρόχου **while** είναι ο ακόλουθος:

```
n=15;
while (n>8)
{
  printf( "%d,",10%n );
  n--;
}
```

3.8. Βρόχοι με συνθήκη εξόδου στη γλώσσα C

3.8.1. Βρόχος do–while

Ο βρόχος **do–while** είναι βρόχος με συνθήκη εξόδου. Η λειτουργία του περιγράφεται εποπτικά στο **Σχήμα 3.4.α** και η σύνταξή του είναι η ακόλουθη:

```
do
{
  Προτάσεις μέσα στις οποίες θα μπορεί να αλλάξει 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 τη λειτουργία του βρόχου
```

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 εκτελεστεί, παρά μετά, αφετέρου δε σε πολλές χρήσεις είναι σημαντικό να μπορεί να παραληφθεί τελείως ο βρόχος εφόσον δεν ικανοποιείται εξαρχής ο έλεγχος. Επιπρόσθετα, για τον βρόχο **do–while** ισχύει η ανάλυση της §3.6.2.

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

3.8.1.1. Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου τμήματος κώδικα:

```
int counter=10;
int counterLimit=20;
do
{
  count++;
  printf( "counter is %d\n",counter );
} while (counter<counterLimit);
<επόμενη πρόταση>;
```

Στις πρώτες δύο γραμμές δηλώνονται και αρχικοποιούνται οι ακέραιες μεταβλητές **counter** και **counterLimit**. Ακολουθεί η πρόταση επανάληψης **do-while**, στην οποία εισέρχεται ο έλεγχος του προγράμματος και εκτελείται η πρώτη επανάληψη. Τυπώνεται

counter is 11

και, ακολούθως, γίνεται ο έλεγχος της συνθήκης: κατά πόσον η μεταβλητή **counter** είναι μικρότερη της **counterLimit**, γεγονός που αληθεύει. Κατά συνέπεια, η επαναληπτική πρόταση συνεχίζεται, έως ότου ο μετρητής καταστεί ίσος με **20**. Για **counter=20** θα εκτελεστεί η επανάληψη και στο τέλος της η συνθήκη θα έχει καταστεί ψευδής, η επαναληπτική πρόταση θα τερματιστεί και ο έλεγχος θα προχωρήσει στην επόμενη πρόταση.

Ο βρόχος εκτελείται συνολικά **10** φορές και δίνει τα αποτελέσματα:

counter is 11
counter is 12
....
counter is 20

Θα πρέπει να σημειωθεί ότι, εάν η συνθήκη ήταν εξαρχής ψευδής (π.χ. **counter=22**), ο βρόχος θα εκτελείτο μία φορά.

3.8.1.2. Παράδειγμα

Να μετασχηματιστεί το τμήμα κώδικα του παραδείγματος 3.7.4.1 σε ισοδύναμο κώδικα με χρήση **do-while**.

```
n=15;
do
{
  printf( "%d,",10%n );
  n--;
} while (n>8);
```

3.8.1.3. Παράδειγμα

Στο πρόγραμμα που ακολουθεί απεικονίζεται η λειτουργία του βρόχου **do-while**, όταν η συνθήκη είναι εξαρχής ψευδής:

```
#include <stdio.h>
int main()
{
   int iteration=0,count=516,limit=40;
    do
   {
      count++;
      iteration++;
      printf( "Inside loop:  ");
      printf( "iteration=%d   count=%d\n",iteration,count);
   } while (count<limit);
   printf("Out of loop: count=%d\n",count);

   return 0;
}
```

Εικόνα 3.7 *Η έξοδος του προγράμματος του παραδείγματος 3.8.1.3*

3.9. Ένθετοι βρόχοι

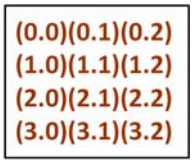
Ένθετος ή εμφωλευμένος ή φωλιασμένος βρόχος (nested loop) ονομάζεται ο βρόχος που περικλείεται σε έναν άλλο. Ο εσωτερικός βρόχος λογίζεται ως μία πρόταση του εξωτερικού βρόχου, κατά συνέπεια σε κάθε επανάληψη του εξωτερικού βρόχου εκτελούνται όλες οι επαναλήψεις του εσωτερικού βρόχου. Η γλώσσα C δεν 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.

3.9.1. Παράδειγμα

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 έναν διπλό βρόχο. Ο εξωτερικός βρόχος **for** έχει μετρητή τη μεταβλητή **i** και σώμα που αποτελείται από δύο προτάσεις: *α)* τον εσωτερικό βρόχο **for** με μετρητή τη μεταβλητή **j** και *β)* την πρόταση **printf("\n");**. Σε κάθε επανάληψη του εξωτερικού βρόχου θα εκτελούνται όλες οι επαναλήψεις του ένθετου και στη συνέχεια θα εκτελείται η **printf("\n");**. Ο τρόπος λειτουργίας του διπλού βρόχου αποτυπώνεται στα αποτελέσματα:



Εικόνα 3.8 *Η έξοδος του προγράμματος του παραδείγματος 3.9.1*

3.9.2. Παράδειγμα

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

```
#include <stdio.h>
int 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 εσωτερικού βρόχου j */
         printf( "\n" );
    }    /* Τέλος του εξωτερικού βρόχου i */

  return 0;
}
```

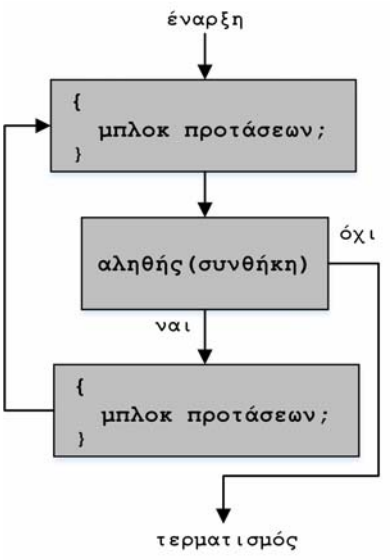
i	i^2	i^3	i^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561

Εικόνα 3.9 *Η έξοδος του προγράμματος του παραδείγματος 3.9.2*

3.10. Διακοπτόμενοι βρόχοι στη γλώσσα C

3.10.1. 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 τον βρόχο. Εάν η **break** βρίσκεται μέσα σε ένθετο βρόχο, τότε επηρεάζεται μόνο ο εσώτερος βρόχος. Η λειτουργία της **break** περιγράφεται εποπτικά στο **Σχήμα 3.5**:



Σχήμα 3.5 *Μπλοκ διάγραμμα της break*

3.10.1.1. Παράδειγμα

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>
int main()
{
  int t;
  for (t=0; t<100; t++)
   {
    printf( "%d ",t );
    if (t==10) break;
  }

return 0;
}
```

3.10.1.2. Παράδειγμα

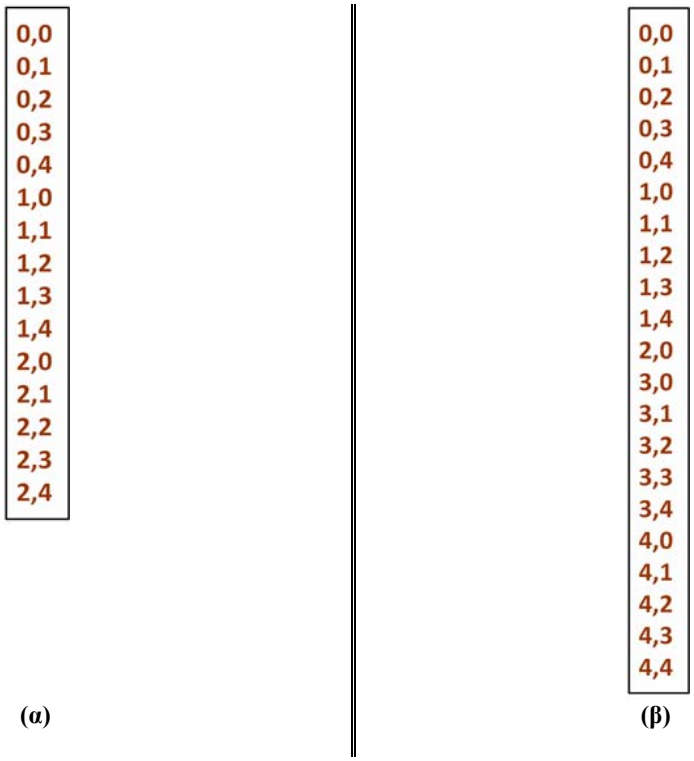
Στην περίπτωση ένθετων βρόχων, η λειτουργία της **break** συναρτάται από τον βρόχο, μέσα στον οποίο βρίσκεται. Στα δύο προγράμματα που ακολουθούν, η διαφορετική τοποθέτηση της **break** οδηγεί σε ριζικά διαφορετικά αποτελέσματα.

```
int main()
{
  int t,p;
  for (t=0; t<5; t++)
  {
    for (p=0;p<5;p++)
    {
      printf( "%d,%d\n",t,p );
    }
    if (t==2) break;
  }

  return 0;
}
```

```
int main()
{
  int t,p;
  for (t=0; t<5; t++)
  {
    for (p=0;p<5;p++)
    {
      printf( "%d,%d\n",t,p );
      if (t==2) break;
    }
  }

  return 0;
}
```

Στο αριστερό πρόγραμμα η **break** βρίσκεται εκτός του εσωτερικού βρόχου και δεν επηρεάζει την εκτέλεσή του. Κατά συνέπεια ο εσωτερικός βρόχος εκτελείται πλήρως για τις πρώτες 3 επαναλήψεις του εξωτερικού βρόχου. Για **t==2** ενεργοποιείται η **break** και διακόπτεται ο εξωτερικός βρόχος.

Στο δεξιό πρόγραμμα η **break** βρίσκεται εντός του εσωτερικού βρόχου, κατά συνέπεια αυτός που θα διακοπεί είναι ο εσωτερικός βρόχος, όταν **t==2** και εκτελείται πλήρως για τις υπόλοιπες τιμές του μετρητή του εξωτερικού βρόχου.



Εικόνα 3.10 *Η έξοδος του προγράμματος του παραδείγματος 3.10.1.2*

3.10.2. Η πρόταση continue

Η λέξη κλειδί **continue** μεταφέρει τον έλεγχο της ροής στην αρχή του βρόχου, παραλείποντας την εκτέλεση του υπόλοιπου τμήματος του σώματος του βρόχου.

Στους βρόχους **while** και **do-while** η **continue** υποχρεώνει τον έλεγχο του προγράμματος να περάσει κατευθείαν στη συνθήκη ελέγχου του βρόχου. Στην περίπτωση του βρόχου **for**, η εκτέλεση του προγράμματος προχωρά στον έλεγχο της τρίτης έκφρασης της δήλωσης **for**.

3.10.2.1. Παράδειγμα

Το παρακάτω πρόγραμμα εμφανίζει στην οθόνη μόνο τους άρτιους αριθμούς.

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

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.

3.10.3. 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** ***πρέπει να αποφεύγεται***, γιατί οδηγεί σε κώδικα «σπαγγέτι» και αίρει τα πλεονεκτήματα του δομημένου προγραμματισμού. Μπορεί να χρησιμοποιηθεί σε περιπτώσεις εξόδου από πολύ βαθιά εμφωλευμένη πρόταση, όπου μία προσεκτική χρήση της **goto** μπορεί να δώσει πιο συμπαγή κώδικα. Θα πρέπει να σημειωθεί ότι η γλώσσα 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.

3.10.3.1. Παράδειγμα

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** δεν 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 της if */
           . . . . . .
     }  /* τέλος της while */
     if (done) break;
  } /* τέλος του εσωτερικού βρόχου for */
  if (done) break;
}  /* τέλος του εξωτερικού βρόχου for */
```

3.11. Κανόνες για τη χρήση των προτάσεων επανάληψης

1. Τοποθετείτε πάντοτε το σώμα των προτάσεων επανάληψης μία θέση στηλοθέτη 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 τον βρόχο επανάληψης συνθήκης εισόδου (**while**) από τον αντίστοιχο συνθήκης εξόδου (**do-while**), γιατί οδηγεί σε πιο ευανάγνωστο κώδικα και προστατεύει από ανεπιθύμητη εκτέλεση μίας επανάληψης στην περίπτωση που η συνθήκη ελέγχου είναι εξαρχής ψευδής.

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. Ελέγχετε σχολαστικά και βεβαιωθείτε ότι κάθε συνθήκη βρόχου επανάληψης οδηγεί στην έξοδο μετά από πεπερασμένες επαναλήψεις, έτσι ώστε να μην δημιουργούνται ατέρμονοι βρόχοι.

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποιος από τους ακόλουθους κανόνες σύνταξης των εντολών switch-case είναι λανθασμένος;
(α) Κάθε **case** πρέπει να έχει μία **int** ή **char** σταθερή έκφραση.
(β) Δύο **case** δεν μπορούν να έχουν την ίδια τιμή.
(γ) Οι προτάσεις κάτω από την ετικέτα **default** εκτελούνται, όταν δεν ικανοποιείται καμία από τις **case** ετικέτες.
(δ) Η **default** είναι απαραίτητα η τελευταία ετικέτα.

(2) Ποιος από τους ακόλουθους κανόνες για τον βρόχο **do-while** είναι λανθασμένος;
(α) Ο βρόχος **do-while** οδηγείται αποκλειστικά από γεγονός.
(β) Ο βρόχος **do-while** ελέγχει τη συνθήκη τερματισμού στο τέλος της εκτέλεσής του.
(γ) Ο βρόχος **do-while** μπορεί να καταστεί λειτουργικά ισοδύναμος με τον βρόχο **for**.
(δ) Ο βρόχος **do-while** αποτελεί επαναληπτική πρόταση.

(3) Ποιες από τις παρατηρήσεις που αφορούν στις ακόλουθες προτάσεις είναι λανθασμένη;

(i)
```
while (++count<12) Πρόταση 1;
```
(ii)
```
while (count++<12) Πρόταση 1;
```

(α) Η πρόταση *(i)* χρησιμοποιεί την προθεματική σημειογραφία, ενώ η *(ii)* τη μεταθεματική.
(β) Στην πρόταση *(i)* πρώτα αυξάνεται η τιμή τής **count** και η νέα τιμή της συγκρίνεται με το **12**.
(γ) Και οι δύο προτάσεις θα εκτελεστούν για τον ίδιο αριθμό επαναλήψεων.
(δ) Στην πρόταση *(ii)* πρώτα συγκρίνεται η τιμή τής **count** με το **12** και στη συνέχεια αυξάνεται η τιμή της.

(4) Ποιες από τις ακόλουθες παραλλαγές του βρόχου **for** είναι λανθασμένη;
(α) Μπορεί να χρησιμοποιηθεί ο τελεστής μείωσης για μέτρηση προς τα κάτω:
```
for (n=10;n>0;n--) printf( "n=%d\n",n );
```
(β) Το βήμα καθορίζεται από τον χρήστη:
```
for (n=0;n<100;n=n+20) printf( "n=%d\n",n );
```

(γ) Χρησιμοποιώντας την ιδιότητα ότι κάθε χαρακτήρας του κώδικα ASCII έχει μία ακέραια τιμή, ο μετρητής μπορεί να είναι μεταβλητή χαρακτήρα:
```
for (n='A';n<'L';n++)
      printf( "n=%d, the ASCII value is %d\n",n,n );
```

(δ) Δεν επιτρέπεται γεωμετρική μεταβολή της τιμής του μετρητή:

```
for (n=1.0;n<100.0;n=2.5*n) printf( "n=%f\n",n );
```

(5) Να μετασχηματιστεί ο βρόχος **for** στον λειτουργικά ισοδύναμο βρόχο **while**:

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

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

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

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

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

Ασκήσεις

Άσκηση 1

Να περιγραφεί η λειτουργία και να δοθούν τα αποτελέσματα του παρακάτω προγράμματος:

```
#include <stdio.h>
int main()
{
    char a='Z';

    while(a>40)
    {
        if (a>'A')  printf("Capital letter:\t");
        printf( "a=%c ASCII value=%d\n",a,a );
        a=a-10;
    }
    printf( "out of loop:  a=%c ASCII value=%d\n",a,a );

    return 0;
}
```

Άσκηση 2

Να γραφεί πρόγραμμα, το οποίο:

***(α)* Θα διαβάζει από το πληκτρολόγιο έναν φυσικό αριθμό (χωρίς έλεγχο απόδοσης ορθής τιμής), τον οποίο θα αποθηκεύει στη μεταβλητή n. Το πρόγραμμα δεν θα επιτρέπει την εισαγωγή αρνητικής τιμής:**

η ανάγνωση του δεδομένου θα γίνεται μέσω επαναληπτικής πρότασης, η οποία θα ζητά από τον χρήστη να δώσει θετική τιμή.
(*β*) Με χρήση της επαναληπτικής πρότασης `for` θα δίνονται από τον χρήστη διαδοχικά `n` ακέραιοι αριθμοί και θα εξαγεται ο μέσος όρος, ο οποίος θα απεικονίζεται στην οθόνη.
(*γ*) Θα υλοποιείται το σκέλος *(β)* με αντικατάσταση της `for` από τη `while` και τη `do-while`.

Άσκηση 3
Να γραφεί πρόγραμμα, το οποίο με χρήση διπλής (ένθετης) επαναληπτικής πρότασης `for` θα απεικονίζει στην οθόνη ολόκληρο τον πίνακα της προπαίδειας (δεν απαιτείται η χρήση πίνακα).

Άσκηση 4
Να γραφεί πρόγραμμα, το οποίο:
***(α)* Με χρήση κατάλληλης επαναληπτικής πρότασης θα διαβάζει από το πληκτρολόγιο χαρακτήρες, έως ότου δοθεί «λευκός» χαρακτήρας, τους οποίους και θα εμφανίζει στην οθόνη (συμπεριλαμβανομένου του «λευκού» χαρακτήρα).**
***(β)* Θα επαναλαμβάνει το σκέλος *(α)* χωρίς εμφάνιση στην οθόνη του «λευκού» χαρακτήρα.**

Άσκηση 5
***(α)* Για το πρόγραμμα που ακολουθεί, να περιγραφεί αναλυτικά η λειτουργία του και να δοθούν τα αποτελέσματά του.**

```
#include <stdio.h>
int main()
{
   float dx=0.2,x,y,sum=0.0,limit_down=0.0,limit_up=0.5;
   for (x=limit_up; x>limit_down; x=x-dx)
   {
      y=5*x-6;
      sum=sum+y*dx;
      printf( "\nx=%f\ty=%6.3f\tsum=%f",x,y,sum );
   }

    return 0;
}
```

***(β)* Να προσαρμοστεί ο ανωτέρω κώδικας, ώστε να καθορίζονται οι τιμές των μεταβλητών `limit_up` και `limit_down` από τον χρήστη και να μετατραπεί ο βρόχος `for` σε ισοδύναμο βρόχο `do-while`.**

Άσκηση 6
Για το πρόγραμμα που ακολουθεί, να περιγραφεί αναλυτικά η λειτουργία του και να δοθούν τα αποτελέσματά του.

```
#include <stdio.h>
int main()
{
   int j;
   float i,x;
   for (i=7.5; i>=1.2; i=i/2.5)
   {
      x=0.0;
      while (x<4)
      {
         printf( "\ni=%f  x=%f",i,x );
         x=x+i;
         do
         {
```

```
            printf( "\nGive an integer greater than 26:" );
            scanf( "%d",&j );
          } while (j<26);
        }
      }

       return 0;
   }
```

Άσκηση 7

Να γραφεί πρόγραμμα, το οποίο θα υπολογίζει τον λογαριασμό ηλεκτρικού ρεύματος, που εκδίδει η επιχείρηση ηλεκτρισμού μίας περιοχής, με βάση τις ακόλουθες προδιαγραφές:

(1) Ο συνολικός λογαριασμός (ΣΛ) προκύπτει ως το άθροισμα ΣΛ=ΛΗΡ+ΛΤ+ΦΠΑ, όπου ΛΗΡ είναι ο λογαριασμός ηλεκτρικού ρεύματος, ΛΤ το σύνολο των λοιπών τελών και ΦΠΑ ο φόρος προστιθέμενης αξίας με συντελεστή 8% επί του ΛΗΡ (ΦΠΑ=8% του ΛΗΡ).

(2) Για τον υπολογισμό του ΛΗΡ ακολουθείται η εξής τιμολογιακή πολιτική για έναν οικιακό καταναλωτή:

Έστω ΚΩ (σε κιλοβατώρες) η συνολική κατανάλωση κατά τη διάρκεια ενός τετραμήνου. Το τίμημα της κατανάλωσης (ΤΚ) προκύπτει ως εξής:

- **Για κατανάλωση από 1 έως 820 κιλοβατώρες: 0.06817 €/ κιλοβατώρα**
- **Για τις επόμενες κιλοβατώρες από 821 έως 1640: 0.08687 €/ κιλοβατώρα**
- **Για τις επόμενες κιλοβατώρες από 1641 έως 2050: 0.10662 €/ κιλοβατώρα**
- **Για τις επόμενες κιλοβατώρες: 0.14127 €/ κιλοβατώρα**

ΛΗΡ=ΤΚ+ΠΧ-ΠΣ-ΕΡ+ΤΑΠΕ

όπου:

- **ΠΧ είναι η πάγια χρέωση, ύψους 36 €.**
- **ΠΣ είναι το ποσό στρογγυλοποίησης του προηγούμενου λογαριασμού.**
- **ΕΡ είναι η αξία του ρεύματος που δόθηκε ως έναντι στον προηγούμενο λογαριασμό.**
- **ΤΑΠΕ είναι το τέλος για τις ανανεώσιμες πηγές ενέργειας, ύψους 3.5 €.**

(3) Τα λοιπά τέλη ΛΤ αποτελούνται από το άθροισμα των τεσσάρων ακόλουθων τελών:

- **Του τέλους υπέρ της κρατικής ραδιοτηλεόρασης, το οποίο συμβολίζεται ως ΤΝΕΡΙΤ, και ισούται με ΤΝΕΡΙΤ=13.10-ΕΝΑΝ, όπου ΕΝΑΝ το ποσό έναντι του προηγούμενου λογαριασμού.**
- **Του δημοτικού τέλους, ύψους 50 €, το οποίο συμβολίζεται ως ΔΤ.**
- **Του δημοτικού φόρου, ύψους 6.18 €, το οποίο συμβολίζεται ως ΔΦ.**
- **Του τέλους ακίνητης περιουσίας, ύψους 6.80 €, το οποίο συμβολίζεται ως ΤΑΠ.**

(4) Ο ΣΛ στρογγυλοποιείται στο ευρώ, δηλαδή εάν το κόστος είναι 143.34€, ο λογαριασμός στρογγυλοποιείται στα 143 €, ενώ κόστος 143.56 € οδηγεί σε λογαριασμό 144 €.

(5) Το πρόγραμμα δέχεται από τον χρήστη τιμές για τις μεταβλητές ΚΩ, ΠΣ, ΕΡ, ΕΝΑΝ.

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2000), *Γλώσσες Προγραμματισμού (Τόμος Δ)*, ΕΑΠ.

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

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

King, K. (2008), *C Programming: A Modern Approach*, 2nd ed., W.W. Norton & Company.

Prinz, P. & Crawford, T. (2005), *C in a Nutshell*, O'Reilly.

4. Συναρτήσεις

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στις έννοιες του αρθρωτού σχεδιασμού και της συνάρτησης. Δίνονται τα δομικά και λειτουργικά χαρακτηριστικά της συνάρτησης (δήλωση, ορισμός, σώμα, λίστα ορισμάτων, επιστρεφόμενη τιμή) και περιγράφεται η διαδικασία της κλήσης κατ' αξία. Μελετώνται οι κατηγορίες μεταβλητών ανάλογα με την εμβέλεια και τη διάρκειά τους και περιγράφονται οι λειτουργίες των τοπικών και static μεταβλητών μέσα στις συναρτήσεις. Αναλύεται η έννοια της αναδρομής και παρουσιάζεται η λειτουργία των αναδρομικών συναρτήσεων με τη βοήθεια παραδειγμάτων. Στο τέλος του κεφαλαίου παρατίθενται δύο παραδείγματα ανάπτυξης δομημένου κώδικα.

Λέξεις κλειδιά
δομημένος προγραμματισμός, συνάρτηση, λίστα ορισμάτων, επιστρεφόμενη τιμή, void, καλούσα και καλούμενη συνάρτηση, κλήση κατ' αξία, αναδρομική συνάρτηση, εμβέλεια και διάρκεια μεταβλητών, τοπική μεταβλητή, καθολική μεταβλητή, static μεταβλητή, αρθρωτός σχεδιασμός

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

4.1. Οι έννοιες του αρθρωτού σχεδιασμού και της συνάρτησης

Ο δομημένος προγραμματισμός στηρίζεται στην έννοια του *αρθρωτού σχεδιασμού* (modular design), δηλαδή στον μερισμό σύνθετων προβλημάτων σε επιμέρους μικρά και απλούστερα τμήματα. Κατ' αντιστοιχία, ένα μεγάλο πρόγραμμα μπορεί να τεμαχιστεί σε μικρότερες γλωσσικές κατασκευές. Στη γλώσσα 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.λπ. Με βάση τις συναρτήσεις αυτές μπορούν να εξαχθούν τα ακόλουθα βασικά χαρακτηριστικά:

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

4.2. 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 4.1** δίνει τη μορφή ενός δομημένου προγράμματος στη γλώσσα C.

Μία συνάρτηση περιλαμβάνει τρεις φάσεις:

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

```
εντολές προεπεξεργαστή (#include, #define, ...)
δηλώσεις συναρτήσεων
δηλώσεις μεταβλητών (επονομαζόμενες καθολικές μεταβλητές)
int main()
{
     δηλώσεις μεταβλητών
     προτάσεις
}
function1()
{
     ......
}
function2()
{
     ......
}
```

Σχήμα 4.1 *Γενική μορφή προγράμματος στη γλώσσα C*

4.2.1. 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. Η ονοματοδοσία των συναρτήσεων ακολουθεί του κανόνες ονοματοδοσίας των μεταβλητών.
- **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 maximumTwoIntegers( int firstInteger, int secondInteger );
```

αποτελεί τη δήλωση μίας συνάρτησης ονόματι `maximumTwoIntegers`, η οποία δέχεται δύο εισόδους, τις ακέραιες μεταβλητές `firstInteger` και `int secondInteger`, και επιστρέφει ως έξοδο έναν ακέραιο (ο τύπος `int` πριν από το όνομά της).

Η δήλωση των συναρτήσεων γίνεται πριν από τη `main()`, συνήθως μετά τις εντολές προεπεξεργαστή.

4.2.2. Ορισμός συνάρτησης

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

```
<τύπος δεδομένων επιστροφής>  <όνομα συναρτησης>
{
     πρόταση;
     ......
     πρόταση επιστροφής;  /* εφόσον η συνάρτηση επιστρέφει τιμή */

}
```

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

```
int maximumTwoIntegers(int firstInteger,int secondInteger)
{
     if (firstInteger>secondInteger) return(firstInteger);
     else return(secondInteger);
}
```

Παρατηρήσεις:

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

```
void printTwoIntegers(int firstInteger,int secondInteger)
{
     if (firstInteger>secondInteger)
       printf( "max=%d\n",firstInteger );
     else printf( "max=%d\n",firstInteger );
}
```

Εάν δεν δηλωθεί μία συνάρτηση ως **void**, τα παλαιότερα πρότυπα της γλώσσας C θεωρούν επιστρεφόμενη τιμή **integer**. Το πρότυπο C99 έχει καταργήσει τη σύμβαση αυτή.

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

4.2.3. Κλήση συνάρτησης

Μία συνάρτηση ενεργοποιείται, όταν κληθεί. Το τμήμα κώδικα απ' όπου καλείται, ονομάζεται **καλούσα** συνάρτηση και μπορεί να είναι είτε η **main()** είτε μία άλλη συνάρτηση. Η συνάρτηση που καλείται ονομάζεται **καλούμενη** συνάρτηση. Εάν η συνάρτηση δεν επιστρέφει τιμή, η κλήση της από ένα σημείο της καλούσας συνάρτησης γίνεται 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 στον 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 **printTwoIntegers()** μπορεί να λάβει την ακόλουθη μορφή:

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

Όταν κληθεί η συνάρτηση, το τυπικό όρισμα **firstInteger** αντιστοιχεί στο πραγματικό όρισμα **x**, οπότε **firstInteger=x=10**, και το τυπικό όρισμα **secondInteger** 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 **maximumTwoIntegers** μπορεί να λάβει την ακόλουθη μορφή:

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

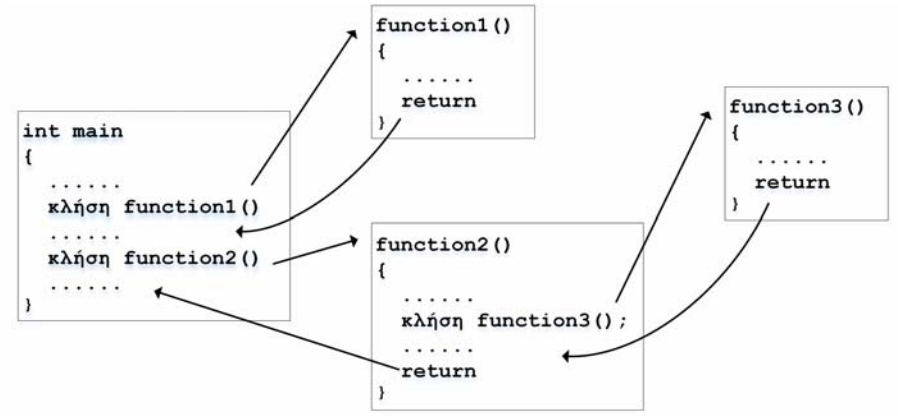
Η διαφορά έγκειται στο ότι στο τέλος της συνάρτησης η επιστρεφόμενη τιμή θα πρέπει να αποδοθεί σε μία μεταβλητή της καλούσας συνάρτησης, στην παρούσα περίπτωση στη μεταβλητή **y**. Κατά συνέπεια, αφενός μεν η **y** θα πρέπει να είναι ίδιου τύπου με την επιστρεφόμενη τιμή (**integer** στο συγκεκριμένο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

Όταν καλείται μία συνάρτηση, π.χ. **y=maximumTwoIntegers(x,32);**, οι τιμές **x** (δηλαδή το **10**) και **32** αντιγράφονται μία προς μία στις μεταβλητές που συνιστούν τη λίστα παραμέτρων. Αυτό σημαίνει ότι η συνάρτηση δεν διαχειρίζεται τα αυθεντικά δεδομένα αλλά αντίγραφα αυτών. Αυτός ο τρόπος κλήσης ονομάζεται **κλήση κατ' αξία** (call by value). Στο Κεφάλαιο 6 θα μελετηθεί ένας δεύτερος τρόπος κλήσης συναρτήσεων, η **κλήση κατ' αναφορά** (call by reference).

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

Ο τρόπος επικοινωνίας ανάμεσα στα τμήματα ενός δομημένου προγράμματος απεικονίζεται στο **Σχήμα 4.2**. Όταν καλείται μία συνάρτηση (είτε από τη **main()** είτε από άλλη συνάρτηση), ο έλεγχος ροής του προγράμματος περνά σε αυτήν και αρχίζει η εκτέλεση των εντολών της. Ο έλεγχος του προγράμματος επιστρέφει στην καλούσα συνάρτηση είτε όταν εκτελεσθούν όλες οι εντολές της καλούμενης συνάρτησης είτε όταν εκτελεσθεί εντολή **return**. Η επιστροφή στην καλούσα συνάρτηση γίνεται στην εντολή που ακολουθεί την κλήση της καλούμενης συνάρτησης.

Κατά συνέπεια, στην πρόταση **y=maximumTwoIntegers(x,32);** πρώτα εκτελείται η εντολή κλήσης συνάρτησης και, όταν ολοκληρωθεί η λειτουργία της **maximumTwoIntegers**, η επιστρεφόμενη τιμή ανατίθεται στη μεταβλητή **y** της **main()**.



Σχήμα 4.2 *Η λειτουργία ενός δομημένου προγράμματος*

Για τη διαχείριση των συναρτήσεων δημιουργείται στη μνήμη του υπολογιστή μία στοίβα κλήσεων (call stack). Κάθε φορά που καλείται μία συνάρτηση, προστίθεται μία εγγραφή στη στοίβα κλήσεων, που περιέχει πληροφορίες σχετικά με την κατάσταση της συνάρτησης, δηλαδή πληροφορίες σχετικά με τις παραμέτρους της συνάρτησης, τις μεταβλητές της, τη διεύθυνση της επιστροφής κ.λπ. Με το πέρας της συνάρτησης η στοίβα αδειάζει από την εγγραφή, δηλαδή οι παράμετροι και οι μεταβλητές της συνάρτησης παύουν να υφίστανται, υπό την έννοια ότι οι θέσεις μνήμης που καταλάμβαναν θεωρούνται πλέον μη σχετιζόμενες με τη λειτουργία του προγράμματος, άρα ελεύθερες προς διάθεση.

4.2.3.1. Παράδειγμα

Στο πρόγραμμα που ακολουθεί παραλήφθηκε η δήλωση της συνάρτησης **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);
}

int main()
{
   float x=15.2;
   printf( "x^2=%f\n",square(x) );

   return 0;
}
```

4.2.3.2. Παράδειγμα

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>

int main()
{
   float degFar,degCel, ratio;
   printf( "Enter degrees F: " );
   scanf( "%f",&degFar );
   /* Κώδικας μετατροπής: */
   ratio=(float)5/9;
   degCel=(degFar-32)*ratio;
   printf( "%f degrees F are %f degrees C\n",degFar,degCel );

   return 0;
}
```

Σε περίπωση που η μετατροπή χρειαζόταν σε πολλά σημεία ενός προγράμματος, δεν θα έπρεπε να επαναλαμβάνεται ο κώδικας της μετατροπής των θερμοκρασιακών κλιμάκων. Για τον 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);

int main()
{
    float degFar,degCel;
    printf( "Enter degrees F: " );
    scanf( "%f",&degFar );
    degCel=F_to_C(degFar);
    printf( "%f degrees F are %f degrees C\n",degFar,degCel );
}   /* τέλος της main */

float F_to_C(float far)
{
    float ratio=5.0/ 9.0;
    return((far-32)*ratio);
} /* τέλος της συνάρτησης */
```

4.2.3.3. Παράδειγμα

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. Ο κώδικας που υλοποιεί την ανωτέρω υπόθεση είναι ο ακόλουθος:

```
#include <...h>
#define ......

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

int main()
{
    int num=5;
    printf( "%d\n",max(12/2,2*num) );
    printf( "%f\n",power(num,3) );

    return 0;
}
```

4.3. Είδη και εμβέλεια μεταβλητών

4.3.1. Τοπικές μεταβλητές

Στο παράδειγμα 4.2.3.2 η συνάρτηση `F_to_C()` περιέχει στο σώμα της τη μεταβλητή `ratio`. Η μεταβλητή αυτή λειτουργεί 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 λειτουργεί (ή αλλιώς είναι προσπελάσιμη ή «ορατή») μέσα στη συνάρτηση που δηλώνεται. Κατ' αντιστοιχία,οι μεταβλητές που δηλώνονται μέσα στη συνάρτηση `main()` δεν είναι προσπελάσιμες από τις υπόλοιπες συναρτήσεις του προγράμματος.

4.3.1.1. Παράδειγμα

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 λειτουργούν ανεξάρτητα η μία από την άλλη.

```
#include<stdio.h>
float square (float x);

int main()
{
   float in,out;
   in=-4.0;
   out=square(in);
   printf( "out within main() equals %.3f\n",out );
   printf( "%.3f squared equals %.3f\n",in,out );

   return 0;
}

float  square (float x)
{
   float out;
   out=24.5;
   printf( "out within square() equals %.3f\n",out );
   return(x*x);
}
```

```
out within square() equals 24.5000
out within main() equals 16.000
-4.000 squared equals 16.000
```

Εικόνα 4.1 *Η έξοδος του προγράμματος του παραδείγματος 4.3.1.1*

4.3.2. Καθολικές μεταβλητές

Σε αντιδιαστολή με τις τοπικές μεταβλητές, οι οποίες είναι προσπελάσιμες από κώδικα που βρίσκεται 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 vari-

ables) και στην περίπτωση που το πρόγραμμα αποτελείται από ένα μόνο αρχείο κώδικα, δηλώνονται εκτός της συνάρτησης **main()**. Όταν μεταβάλλεται η τιμή μίας καθολικής μεταβλητής σε οποιοδήποτε σημείο του προγράμματος, η νέα τιμή είναι ορατή από ολόκληρο 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.

4.3.2.1. Παράδειγμα

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);

int main()
{
   float in;
   glob=2.0;
   printf( "glob within main equals %.3f\n",glob );
   in=square(glob);
   printf( "%.3f squared equals %.3f\n",glob,in );
   in=square(glob);
   printf( "%.3f squared equals %.3f\n",glob,in );

   return 0;
}

float square(float x)
{
   glob=glob+1.0;
   printf( "glob within square equals %.3f\n",glob );
   return (x*x);
}
```

```
glob within main equals 2.000
glob within square equals 3.000
3.000 squared equals 4.000
glob within square equals 4.000
4.000 squared equals 9.000
```

Εικόνα 4.2 *Η έξοδος του προγράμματος του παραδείγματος 4.3.2.1*

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`.

4.3.3. Εμβέλεια μεταβλητών

Στις §4.3.1 και 4.3.2 παρουσιάστηκαν δύο είδη μεταβλητών με διαφορετικό εύρος λειτουργίας. Το τμήμα του πηγαίου κώδικα στο οποίο μία μεταβλητή είναι ορατή 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 x`.
- **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 γλώσσα 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***.

4.3.3.1. Παράδειγμα

Να προσδιοριστεί η εμβέλεια των μεταβλητών στον ακόλουθο πηγαίο κώδικα:

```
1 #include <stdio.h>
2
3 int function1(int x, int y);
4 void function2(int z);
5
6 int x,y;
7
8 int main()
9 {
10      int x=20; y=x--;
11      printf( "x:%d  y:%d  function1(y+5,x):%d\n",x,y,
        function1(y+5,x) );
12      function2(x+y);
13      return 0;
14   }  /* τέλος της main */
15   int w=100;
16   int function1(int x, int y)
17   {
18      return(x>y?x:y);
19   }  /* τέλος της function1 */
20   void function2(int z)
21   {
22      int y=35;
23      printf( "x:%d  y:%d  w:%d  z:%d  function1(z,y):%d\n",
```

```
    x,y,w,z,function1(z,y) );
24  } /* τέλος της function2 */
```

```
x:19 y:20 function(y+5,x):25
x:0 y:35 w:100 z:39 function1(z,y):39
```

Εικόνα 4.3 *Η έξοδος του προγράμματος του παραδείγματος 4.3.3.1*

- `6 int x,y;` Οι μεταβλητές αυτές είναι καθολικές, με εμβέλεια προγράμματος.
- `15 int w=100;` Έχει εμβέλεια προγράμματος αλλά είναι ενεργή από το σημείο δήλωσής της και κάτω (γραμμή 15).
- `3 int function1(int x, int y);` Οι μεταβλητες `x` και `y` έχουν εμβέλεια μπλοκ και αποκρύπτουν για το σώμα της `function1()` τις καθολικές μεταβλητές `x` και `y`.
- `10 int x=20; y=x--;` Η `y` αποκτά την αρχική τιμή της `x` (δηλαδή **20**) και ακολούθως η τιμή της `x` μειώνεται κατά **1**.
- `12 function2(x+y);` Καλείται η συνάρτηση `function2()` και αυτή δίνει στο τυπικό όρισμα `z` την τιμή **19+20=39**. Το όρισμα `z` έχει εμβέλεια μπλοκ. Η τοπική μεταβλητή `y=35`, που δηλώνεται στη γραμμή 22, αποκρύπτει από το σώμα της `function2()` την καθολική μεταβλητή `y`. Δεν συμβαίνει, όμως, το ίδιο και για την καθολική μεταβλητή, η οποία είναι ορατή από το σώμα της `function2()`.
- `23 printf( "x:%d y:%d w:%d z:%d function1(z,y):%d\n", x,y,w,z,function1(z,y) );` Καλείται η συνάρτηση `function1(z,y)` με ορίσματα **39** και **35**, τα οποία αντιγράφονται στις παραμέτρους `x` και `y` της `function1()`. Η `function1()` επιστρέφει το **39**, το οποίο τυπώνει η `printf()`.

4.3.4. Διάρκεια μεταβλητών – static μεταβλητές

Η διάρκεια καθορίζει τον χρόνο κατά τον οποίο το όνομα μίας 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 αυτήν, αντίστοιχα.

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

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

Εάν προστεθεί στη δήλωση μίας τοπικής μεταβλητής η λέξη `static`, τότε η μεταβλητή διατηρεί την τιμή της και καθίσταται πλήρους διάρκειας. Αυτό σημαίνει ότι μία `static` μεταβλητή παρουσιάζει τα ακόλουθα χαρακτηριστικά:

- Κατά την πρώτη κλήση της συνάρτησης στην οποία βρίσκεται, μία `static` μεταβλητή δημιουργείται και– κατά την εκτέλεση της συνάρτησης– λαμβάνει τιμή. Όταν ολοκληρώνεται η πρώτη κλήση της συνάρτησης, η μεταβλητή και η τιμή που απέκτησε διατηρούνται εν ενεργεία.
- Σε κάθε επόμενη κλήση της συνάρτησης δεν εκτελείται η πρόταση δήλωσης της `static` μεταβλητής, αλλά η μεταβλητή υφίσταται και μάλιστα με την τιμή που είχε, όταν ολοκληρώθηκε η προηγούμενη κλήση της συνάρτησης. Μετά το πέρας της κλήσης της συνάρτησης, η μεταβλητή διατηρείται με τη νέα τιμή που – πιθανόν – έλαβε στην κλήση αυτή. Αυτή η διαδικασία συνεχίζεται καθόλη τη διάρκεια εκτέλεσης του προγράμματος.

Στη συνάρτηση

```
func(int x)
{
```

```
        int temp;
        static int num;
        . . . . . .
    }
```

η μεταβλητή **num** είναι τοπική, αλλά ως **static** 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 οριστεί να γίνεται, με κάθε είσοδο στο μπλοκ που αυτή ορίζεται. Αντίθετα, μία τοπική μεταβλητή πλήρους διάρκειας αρχικοποιείται μόνο με την ενεργοποίηση του προγράμματος.

4.3.4.1. Παράδειγμα

Ο κώδικας που ακολουθεί, αποτελεί παράδειγμα χρήσης στατικών μεταβλητών. Υλοποιεί τον υπολογισμό του κυλιόμενου μέσου όρου, σύμφωνα με τον οποίο προστίθενται διαδοχικά δεδομένα και σε κάθε προσθήκη δεδομένου υπολογίζεται ο νέος μέσος όρος. Με αυτόν τον τρόπο δημιουργείται μία ακολουθία μέσων όρων ως εξής:

- Η εισαγωγή του πρώτου δεδομένου x_1 οδηγεί στον μέσο όρο $MO_1 = \frac{x_1}{1}$
- Η εισαγωγή του δεύτερου δεδομένου x_2 οδηγεί στον μέσο όρο $MO_2 = \frac{x_1 + x_2}{2} = \frac{x_1 + x_2}{1+1}$
- Η εισαγωγή του τρίτου δεδομένου x_3 οδηγεί στον μέσο όρο $MO_3 = \frac{x_1 + x_2 + x_3}{2} = \frac{(x_1 + x_2) + x_3}{2+1}$
- Η εισαγωγή του k-στου δεδομένου x_k οδηγεί στον μέσο όρο $MO_k = \frac{x_1 + x_2 + \ldots + x_k}{k} = \frac{(x_1 + x_2 + \ldots + x_{k-1}) + x_k}{(k-1)+1}$

Με βάση τα παραπάνω, αν ορίσουμε $MO_k = \frac{A_k}{\Pi_k}$, ο υπολογισμός του μέσου όρου μετά την εισαγωγή του επόμενου δεδομένου προκύπτει από τη σχέση: $MO_{k+1} = \frac{A_k + x_{k+1}}{\Pi_k + (k+1)}$, δηλαδή μπορεί να εξαχθεί βάσει των ήδη υπολογισθεισών τιμών των A_k και Π_k.

Η υλοποίηση των ανωτέρω γίνεται στο πρόγραμμα που ακολουθεί. Στη συνάρτηση **getAverage()** οι μεταβλητές **static float total** και **static int count** δηλώνονται και αρχικοποιούνται μόνο την πρώτη φορά. Στις επόμενες κλήσεις διατηρούν τα αποτελέσματα της προηγούμενης κλήσης και σε αυτά προστίθενται στη μεν **total** το **newdata**, στη δε **count** η μονάδα.

```
#include <stdio.h>

float getAverage(float newdata);

int main()
{
   float data=1.0;
   float average;
   while (data!=0)
   {
      printf( "\n Give a number or press 0 to finish: " );
```

```
        scanf( "%f",&data );
        average=getAverage(data);
        printf( "The new average is %f\.3n",average );
    }
} /* τέλος της main */

float getAverage(float newdata)
{
    static float total=0.0;
    static int count=0;
    count++;
    total=total+newdata;
    return(total/count);
} /* τέλος της get_average */
```

```
Give a number or press 0 to finish: 10
The new average is 10.000
Give a number or press 0 to finish: 38
The new average is 24.000
Give a number or press 0 to finish: 72
The new average is 40.000
Give a number or press 0 to finish: 67.5
The new average is 46.875
```

Εικόνα 4.4 *Η έξοδος του προγράμματος του παραδείγματος 4.3.4.1*

Είναι προφανές ότι η έννοια της **static** μεταβλητής είναι σαφέστατα πιο σύνθετη από εκείνη της απλής τοπικής μεταβλητής. Η υλοποίηση, όμως, ορισμένων προγραμμάτων με χρήση **static** μεταβλητών μπορεί να επιφέρει σημαντικές βελτιώσεις στις απαιτήσεις αποθηκευτικού χώρου και στον χρόνο εκτέλεσης ενός προγράμματος. Στο συγκεκριμένο παράδειγμα, εάν χρησιμοποιείτο προσέγγιση χωρίς **static** μεταβλητές, θα έπρεπε σε κάθε επανάληψη να αθροίζονται όλα τα δεδομένα που εισήχθησαν στις προηγούμενες επαναλήψεις, τα οποία μάλιστα θα έπρεπε να παραμείνουν αποθηκευμένα στη μνήμη. Αυτό σημαίνει ότι στην *k*-στη επανάληψη για τον υπολογισμό του αριθμητή του μέσου όρου θα έπρεπε να γίνουν (**k-1**) προσθέσεις. Κατά συνέπεια, η διαδοχική εισαγωγή ***N*** δεδομένων θα απαιτούσε ***N*** θέσεις μνήμης για την αποθήκευση των δεδομένων και $1+2+...+(\boldsymbol{N}-1)=\frac{(\boldsymbol{N}-1)\cdot(\boldsymbol{N}-2)}{2}$ προσθέσεις, δηλαδή η υλοποίηση του κυλιόμενου μέσου όρου θα απαιτούσε αλγόριθμο τάξης $\boldsymbol{N}^2$.

Αντίθετα, η χρήση **static** μεταβλητών απαιτεί μόνο 2 θέσεις μνήμης (μία για τον αριθμητή και μία για τον παρονομαστή). Επιπρόσθετα, σε κάθε νέα εισαγωγή δεδομένου γίνεται μία πρόσθεση για τον υπολογισμό του αριθμητή και μία ακόμη για τον υπολογισμό του παρονομαστή. Επομένως, η διαδοχική εισαγωγή ***N*** δεδομένων θα απαιτούσε ***2N*** προσθέσεις, δηλαδή η υλοποίηση του κυλιόμενου μέσου όρου θα απαιτούσε αλγόριθμο τάξης ***N***.

4.4. Αναδρομικές συναρτήσεις

Μία συνάρτηση ονομάζεται αναδρομική, όταν σε μία πρόταση του σώματός της η συνάρτηση καλείται εκ νέου. Η αναδρομή είναι μία διαδικασία με την οποία ορίζεται κάτι μέσω του ίδιου του οριζόμενου.

Έστω ότι ένα πρόβλημα επιχειρείται να επιλυθεί με χρήση αναδρομικής συνάρτησης. 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 αποσαφηνιστεί με τη βοήθεια του προβήματος υπολογισμού του παραγοντικού ενός ακέραιου αριθμού $\mathbf{n}!=1\cdot 2\cdot ...\cdot \mathbf{n}$. Ως βάση της αναδρομής θεωρείται η ισότητα $0!=1!=1$. Στους παρακάτω κώδικες θεωρείται ότι δίνεται φυσικός αριθμός και δεν απαιτείται έλεγχος τιμής:

```
/* Μη αναδρομική εκδοχή */
int fact(int n)
{
  int i,total=1;
  if ((n==0) || (n==1))
    return(1);
  else
  {
    for (i=2; i<=n; i++)
      total=total*i;
  }
  return(total);
}
```

```
/* Αναδρομική εκδοχή */
int fact(int n)
{
  if ((n==0) || (n==1))
    return(1);
  else return(fact(n-1)*n);
}
```

Επεξηγήσεις της αναδρομικής εκδοχής:

```
if ((n==0) || (n==1)) return(1);
else return(fact(n-1)*n);
```

Εάν το **n** είναι ίσο με **0** ή **1**, τότε το παραγοντικό είναι **1**. Στη γενική περίπτωση, ο υπολογισμός του παραγοντικού του αριθμού `n` μπορεί να θεωρηθεί ως υπολογισμός του γινομένου $1\cdot 2\cdot ...\cdot(\mathbf{n}-1)$ επί το **n**, δηλαδή $((\mathbf{n}-1)!)\cdot\mathbf{n}$. Αντίστοιχα, ο υπολογισμός του παραγοντικού $(\mathbf{n}-1)!$ δίνεται από τη σχέση $(\mathbf{n}-1)!=((\mathbf{n}-2)!)\cdot(\mathbf{n}-1)$. Ακολουθώντας τη διαδικασία αυτή, μπορούμε να ορίσουμε τα εξής:

$$\mathbf{n}!=(\mathbf{n}-1)!\cdot\mathbf{n}$$
$$(\mathbf{n}-1)!=((\mathbf{n}-2)!)\cdot(\mathbf{n}-1)$$
$$(\mathbf{n}-2)!=((\mathbf{n}-3)!)\cdot(\mathbf{n}-2)$$
$$(\mathbf{n}-3)!=((\mathbf{n}-4)!)\cdot(\mathbf{n}-3)$$

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 συνάρτηση `fact()` με όρισμα `n`. Με την πρόταση `return(fact(n-1)*n)` η `fact()` 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 συνεχιστεί, έως ότου προκληθεί διακοπή των κλήσεων. Η διακοπή είναι αποκλειστική ευθύνη του προγραμματιστή. Στο συγκεκριμένο παράδειγμα η διακοπή προκαλείται με την πρόταση `if ((n==0) || (n==1))`, που σημαίνει ότι, όταν το `n` φτάσει να γίνει **1**, υπάρχει πλέον αποτέλεσμα. Έτσι, οι διαδοχικές κλήσεις για `n=4` είναι:

fact(4) καλεί τη fact(3)
fact(3) καλεί τη fact(2)
fact(2) καλεί τη fact(1)
η fact(1) δίνει αποτέλεσμα 1 και το επιστρέφει στη fact(2)
η fact(2) δίνει αποτέλεσμα 1*2=2 και το επιστρέφει στη fact(3)
η fact(3) δίνει αποτέλεσμα 2*3=6 και το επιστρέφει στη fact(4)
η fact(4) δίνει αποτέλεσμα 6*4=24, το οποίο είναι και το τελικό

Στο πρόγραμμα που ακολουθεί υλοποιείται η ανωτέρω αναδρομική συνάρτηση. Επιπρόσθετα, έχει εισαχθεί ο μετρητής `numberOfCalls`, ο οποίος καταγράφει τις κλήσεις στη συνάρτηση `fact()`. Ο μεταβλητής υλοποιείται με καθολική μεταβλητή, για να είναι προσβάσιμος από οποιοδήποτε σημείο του κώδικα.

```c
#include <stdio.h>

int sum(int n);

int numberOfCalls=0;

int main()
{
    int n;
    do
    {
       printf( "\nGive a natural number: " );
       scanf("%d",&n );
     } while (n<0);
    printf( "\n Factorial = %d", fact(n) );

  return 0;
} /* Τέλος της main */

int fact(int n)
{
    if ((n==0) || (n==1))
    {
       numberOfCalls++;
       printf( "\nNumber of calls:%d",numberOfCalls );
       return(1);
    }
    else
    {
       numberOfCalls++;
       printf( "\nNumber of calls:%d",numberOfCalls );
       return(fact(n-1)*n);
    }
} /* Τέλος της fact */
```

```
Give a natural number: 4

Number of calls:1
Number of calls:2
Number of calls:3
Number of calls:4
 Factorial = 24
```

Εικόνα 4.5 *Η έξοδος του προγράμματος υπολογισμού του παραγοντικού*

Πλεονεκτήματα των αναδρομικών συναρτήσεων:

- Το πλέον βασικό πλεονέκτημα των αναδρομικών συναρτήσεων είναι ότι αποτελούν υλοποιήσεις αναδρομικών αλγορίθμων, όπως οι μέθοδοι ταξινόμησης δεδομένων.
- 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.
- Όταν σε μία συνάρτηση γίνονται πολλές αναδρομικές κλήσεις, υπάρχει πιθανότητα να προκληθεί 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.

4.4.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα υλοποιεί τον τύπο του Euler για τον προσεγγιστικό υπολογισμό του **π**. Το πρόγραμμα θα δέχεται από το πληκτρολόγιο έναν φυσικό αριθμό ***k*** και θα καλεί την αναδρομική συνάρτηση **`double calcPi(int k)`**. Η συνάρτηση θα δέχεται ως είσοδο το ***k*** και θα επιστρέφει στη **`main()`** το άθροισμα $\sum_{n=1}^{k}\frac{1}{n^2}$. Ακολούθως, μέσα στη **`main()`** θα εκτελείται η πράξη $\sqrt{6\cdot\sum_{n=1}^{k}\frac{1}{n^2}}$, η οποία οδηγεί στην προσεγγιστική τιμή του **π**.

```
#include <stdio.h>
#include <math.h> /* για τη συνάρτηση sqrt */

double calcPi(int k);

int main()  {
   int k;
   double p;
   do
   {
    printf( "\nGive a big positive integer:  " );
      scanf("%d",&k);
    } while (k<0);
    p=sqrt(6*calcPi(k));
    printf( "\nThe approximate value of pi is %lf",p );
}

double calcPi(int k)
{
   double sum;
   if (k==1) return(1.0);
   else
   {
       sum=(1.0/(k*k))+calcPi(k-1);
    return(sum);
    }
}
```

```
Give a big positive integer:  57900

The approximate value of pi is 3.141565
```

Εικόνα 4.6 *Η έξοδος του προγράμματος του παραδείγματος 4.4.1*

Παρατήρηση: Για τιμές μεγαλύτερες από ένα όριο, εξαρτώμενο από τα χαρακτηριστικά του υπολογιστικού συστήματος στο οποίο εκτελείται ο κώδικας, η αναδρομική συνάρτηση δεν μπορεί να λειτουργήσει, καθώς σημειώνεται υπερχείλιση της στοίβας.

4.5. Αρθρωτός σχεδιασμός

Όπως σημειώθηκε στην αρχή του κεφαλαίου, στόχος του Διαδικαστικού Προγραμματισμού είναι ο αρθρωτός σχεδιασμός, δηλαδή ο μερισμός του συνολικού προβλήματος σε υποπροβλήματα και η επίλυση του καθενός εξ αυτών με αυτόνομες μονάδες κώδικα. Οι συναρτήσεις περιέχουν τα εργαλεία για την υλοποίηση του στόχου αυτού, επιδιώκοντας ταυτόχρονα την επαναχρησιμοποίηση υφιστάμενου κώδικα. Τα δύο παραδείγματα που ακολουθούν, χρησιμοποιούνται ως πρωτόλεια υποδείγματα αρθρωτού σχεδιασμού. Απώτερος στόχος της συγκεκριμένης πρακτικής είναι να καταστεί η συνάρτηση `main()` συντονιστική συνάρτηση, η οποία θα κατανέμει το προς εκτέλεση έργο στα υπόλοιπα τμήματα κώδικα.

4.5.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα διαβάζει χαρακτήρες από το πληκτρολόγιο, θα τους εμφανίζει στην οθόνη και θα τυπώνει το πλήθος των προτάσεων, το πλήθος των λέξεων και το πλήθος των χαρακτήρων του κειμένου. Η ανάγνωση χαρακτήρων θα περατώνεται, όταν δοθεί ο χαρακτήρας του δολαρίου `'$'`.

- Μία πρόταση ολοκληρώνεται όταν αναγνωστεί ένας εκ των χαρακτήρων `'.'` ή `';'` ή `'!'`.
- Μία λέξη ολοκληρώνεται όταν αναγνωστεί ένας εκ των χαρακτήρων `'.'` ή `';'` ή `'!'` ή `'!'` ή `' '` ή `','`.

Σημείωση: Θεωρείται ότι δεν υπάρχουν διαδοχικές εμφανίσεις των χαρακτήρων `'.'` ή `';'` ή `'!'` ή `'!'` ή `' '` ή `','`.

Με βάση τις ανωτέρω προδιαγραφές, το πρόβλημα μπορεί να μεριστεί στα ακόλουθα υποπροβλήματα:

- ανάγνωση χαρακτήρων μέσω επαναληπτικής πρότασης,
- έλεγχος της φύσης του χαρακτήρα και συνακόλουθος χαρακτηρισμός του ως δηλωτικός είτε του τέλους μίας λέξης είτε του τέλους μίας πρότασης,
- εμφάνιση των αποτελεσμάτων στην οθόνη.

Η ανάγνωση των χαρακτήρων θα γίνεται μέσα στη `main()`. Επιπλέον από την κύρια συνάρτηση θα καλούνται οι ακόλουθες συναρτήσεις:

(α) `int endofSentence(char ch)`, η οποία θα δέχεται ως όρισμα έναν χαρακτήρα και θα επιστρέφει **1**, αν ο χαρακτήρας είναι δηλωτικός του τέλους πρότασης, αλλιώς θα επιστρέφει **0**.

(β) `int endofWord(char ch)`, η οποία θα δέχεται ως όρισμα έναν χαρακτήρα και θα επιστρέφει **1**, αν ο χαρακτήρας είναι δηλωτικός του τέλους λέξης, αλλιώς θα επιστρέφει **0**.

(γ) `void displayResults(int s, int w, int c)`, η οποία θα δέχεται τα πλήθη των προτάσεων, λέξεων και χαρακτήρων και θα τα εμφανίζει στην οθόνη.

```
#include <stdio.h>

int endofSentence(char ch);
int endofWord(char ch);
void displayResults(int s, int w, int c);

int main()
{
  FILE *f1;
```

```
    int s=0,w=0,c=0;
    char ch;
    while ((ch=getchar())!='$')
    {
      putchar(ch);
      s=s+endofSentence(ch);
      w=w+endofWord(ch);
      c++;
    }
    displayResults(s,w,c);

    return 0;
  }

  int endofSentence(char ch)
  {
    if ((ch=='.') || (ch==';') || (ch=='!')) return(1);
    else return(0);
  }

  int endofWord(char ch)
  {
    if ((ch=='.') || (ch==';') || (ch=='!') || (ch==' ') || (ch==','))
return(1);
    else return(0);
  }

  void displayResults(int s, int w, int c)
  {
    printf( "\n\nNumber of sentences:%d\nNumber of words:%d\nNumber of
characters:%d",s,w,c );
  }
```

```
Type a number of characters!
Press USD to finish.
$

Number of sentences: 2
Number of words:9
Number of characters:50
```

Εικόνα 4.7 *Η έξοδος του προγράμματος του παραδείγματος 4.5.1*

Όπως προκύπτει από τα αποτελέσματα, ο χαρακτήρας δεν υπολογίζεται στους αναγνωσθέντες. Ο αριθμός των αναγνωσθέντων χαρακτήρων περιλαμβάνει τις δύο αλλαγές γραμμής.

4.5.2. Παράδειγμα

Στο παράδειγμα 4.5.1 η `main()` κατένειμε τμήμα του έργου σε τρεις συναρτήσεις, οι οποίες δεν είχαν επικοινωνία μεταξύ τους. Στο παράδειγμα που ακολουθεί, θα χρησιμοποιηθεί κώδικας που αναπτύχθηκε σε προηγούμενα παραδείγματα (οι συναρτήσεις υπολογισμού του παραγοντικού και υπολογισμού του **π**).

Στόχος είναι να γραφεί πρόγραμμα που θα δέχεται έναν αριθμό μοιρών γωνίας και θα υπολογίζει το ημίτονο και το συνημιτονό της, χρησιμοποιώντας όχι τις διαθέσιμες συναρτήσεις `sin()` και `cos()` αλλά τα

αντίστοιχα αναπτύγματα Taylor. Στα αναπτύγματα Taylor ο χρήστης θα προσδιορίζει τον αριθμό των όρων **n** που θα ενταχθούν, ρυθμίζοντας με αυτόν τον τρόπο την ακρίβεια προσέγγισης των τριγωνομετρικών τιμών.

Τα αναπτύγματα Taylor για τον υπολογισμό του ημιτόνου και του συνημιτόνου δίνονται από τις σχέσεις:

$$\sin(x)=\sum_{i=0}^{n}\frac{(-1)^i}{(2i+1)!}x^{2i+1}=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}\ldots$$

$$\cos(x)=\sum_{i=0}^{n}\frac{(-1)^i}{(2i)!}x^{2i}=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}\ldots$$

Το πρόγραμμα:

- Διαβάζει από το πληκτρολόγιο τη γωνία, **deg**, σε μοίρες και τη μετατρέπει σε ακτίνια από τον τύπο **x=pi*deg/180**.
- Καλεί τις δύο συναρτήσεις **double calcSin(double x, int n)** και **double calcCos(double x, int n)**, οι οποίες επιστρέφουν τις τριγωνομετρικές τιμές. Η παράμετρος **n** στις δύο συναρτήσεις είναι ο αριθμός των όρων που θα χρησιμοποιηθούν στα αναπτύγματα Taylor.
- Ο υπολογισμός του **π** επιτελείται από τη συνάρτηση **double calcPi(int k)**). Για τον υπολογισμό του παραγοντικού θα χρησιμοποιηθεί η συνάρτηση **fact()** της §4.4.
- Ο υπολογισμός των δυνάμεων επιτελείται από την πρότυπη συνάρτηση **pow(x,k)**, η οποία βρίσκεται στο αρχείο κεφαλίδας **math.h** και υλοποιεί την ύψωση του **x** στην ακέραια δύναμη **k**.

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

double calcPi(int k);
int fact(int n);
double calcSin(double x, int n);
double calcCos(double x, int n);

int main()
{
  double imit,sinim,deg,x;
  int n;

  printf("Give an angle (in degrees): ");
  scanf("%lf",&deg);
  printf("\nGive the number of terms in Taylor series: ");
  scanf("%d",&n);

  x=calcPi(57000)*deg/180;
  imit=calcSin(x,n);
  sinim=calcCos(x,n);

  printf("\nsin(%.2lf)=%lf,  cos(%.2lf)=%lf",deg,imit,deg,sinim);

  return 0;
}

double calcPi(int k)
{
  double sum;
  if (k==1) return(1.0);
  else
```

```
  {
       sum=(1.0/(k*k))+calcPi(k-1);
    return(sum);
  }
}

int fact(int n)
{
  if ((n==0) || (n==1))
  return(1);
  else return(fact(n-1)*n);
}

double calcSin(double x, int n)
{
  int i;
  double y=0.0;
  for (i=0;i<=n;i++)
  {
    if (i==0) y=y+x;
    else y=y+(pow(-1,i)*pow(x,(2*i)+1)/fact((2*i)+1));
  }
  return y;
}

double calcCos(double x, int n)
{
  int i;
  double y=0.0;
  for (i=0;i<=n;i++)
  {
    if (i==0) y=y+1;
    else y=y+(pow(-1,i)*pow(x,2*i)/fact(2*i));
     /* κλήση της συνάρτησης pow (πρότυπης) και
   της δημιουργηθείσας συνάρτησης fact */
  }
  return y;
}
```

```
Give an angle (in degrees): 180

Give the number of terms in Taylor series: 4

sin(180.00)=0.997261,  cos(180.00)=-0.074002
```

Εικόνα 4.7 *Η έξοδος του προγράμματος του παραδείγματος 4.5.2*

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;

(α) Μέσα από τη **`main()`** μπορεί να κληθεί οποιαδήποτε συνάρτηση.
(β) Οι συναρτήσεις πρέπει να δηλώνονται, προτού χρησιμοποιηθούν.
(γ) Μία συνάρτηση έχει πάντοτε ένα σώμα, ένα σύνολο προτάσεων και μεταβλητών.
(δ) Μία συνάρτηση έχει πάντοτε εισόδους, μία λίστα ορισμάτων.

(2) Ποιες μεταβλητές έχουν εμβέλεια προγράμματος;
(α) Οι καθολικές.
(β) Οι τοπικές.
(γ) Οι μεταβλητές που δηλώνονται με την λέξη κλειδί **`static`** πριν από τον τύπο τους.
(δ) Δεν υπάρχουν μεταβλητές με εμβέλεια προγράμματος στη γλώσσα C.

(3) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;
(α) Η εμβέλεια μίας στατικής μεταβλητής είναι όση και μίας καθολικής μεταβλητής.
(β) Η παράμετρος μίας συνάρτησης αποτελεί και τοπική μεταβλητή αυτής.
(γ) Οι στατικές μεταβλητές διατηρούν την τιμή τους ανάμεσα στις κλήσεις μίας συνάρτησης.
(δ) Μόλις ολοκληρωθεί η κλήση μίας συνάρτησης, οι τοπικές μεταβλητές αυτής χάνουν τα περιεχόμενά τους.

(4) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;
(α) Για να χρησιμοποιηθεί μία συνάρτηση βιβλιοθήκης, πρέπει στον κώδικα του προγράμματος να συμπεριληφθεί με **`#include`** το αρχείο κεφαλίδας, στο οποίο δηλώνεται.
(β) Εάν μία συνάρτηση δεν έχει τη λέξη κλειδί **`return`**, δεν επιστρέφει ποτέ στο πρόγραμμα που την κάλεσε.
(γ) Μία συνάρτηση στη γλώσσα C δεν μπορεί να έχει περισσότερες της μιας επιστρεφόμενες τιμές.
(δ) Μία συνάρτηση που δηλώνεται πριν από τη **`main()`** μπορεί να οριστεί μετά από τη **`main()`**.

Ασκήσεις

Άσκηση 1

Να περιγραφεί η λειτουργία των παρακάτω προγραμμάτων:

```c
#include <stdio.h>

float func(float x, int n)
{
  static float p = x;
  while ((n-1)>0)
  {
    p=p*x;
    n--;
  }
  return (p);
}

int main()
{
  int i;
  for (i=2;i<6;i++)
   printf("%.2f\n",func(2.0,i));

  return 0;
}
```

```c
#include <stdio.h>

float func(float x, int n)
{
  float p = x;
  while ((n-1)>0)
  {
    p=p*x;
    n--;
  }
  return (p);
}

int main()
{
  int i;
  for (i=2;i<6;i++)
  printf("%.2f\n",func(2.0,i));

  return 0;
}
```

Άσκηση 2

Να περιγραφεί η λειτουργία του παρακάτω προγράμματος:

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

```
float pown(float x, int n)
{
   float p = x;
   while (n-1)
   {
       p=p*x;
      n--;
   }
   return (p);
}

int main()
{
   printf( "%.2f",pown(2.0,10) );

   return 0;
}
```

Άσκηση 3
Να γραφεί συνάρτηση, η οποία θα δέχεται ως όρισμα τη βαθμολογία ενός φοιτητή και θα επιστρέφει έναν από τους ακόλουθους χαρακτήρες:
`'E'`, εάν ο βαθμός κείται στο διάστημα [8.5,10]
`'V'`, εάν ο βαθμός κείται στο διάστημα [6.5,10)
`'G'`, εάν ο βαθμός κείται στο διάστημα [5.0,6.5)
`'F'`, εάν ο βαθμός είναι μικρότερος του 5.0

Άσκηση 4
Να γραφεί συνάρτηση, η οποία θα δέχεται ορίσματα τρεις ακέραιους αριθμούς και θα επιστρέφει τον ενδιάμεσο αυτών.

Άσκηση 5
Να γραφεί πρόγραμμα, το οποίο θα δέχεται από τον χρήστη χρονολογίες (π.χ. 1861) και θα εμφανίζει κατάλληλο μήνυμα που θα πληροφορεί κατά πόσον το έτος είναι δίσεκτο. Προς τούτο να χρησιμοποιηθεί ο αλγόριθμος υπολογισμού των δίσεκτων ετών, που αναπτύχθηκε στο παράδειγμα 2.1 του κεφαλαίου 3.

Άσκηση 6
Να γραφεί πρόγραμμα, το οποίο θα δέχεται την επιλογή του χρήστη για το γεωμετρικό σχήμα που επιθυμεί και θα υπολογίζει το εμβαδό του. Προς τούτο θα δημιουργηθούν τρεις συναρτήσεις, οι οποίες θα δέχονται τις διαστάσεις του τετραγώνου, του ορθογωνίου παραλληλογράμμου και του κύκλου και θα επιστρέφουν το αντίστοιχο εμβαδό.

Άσκηση 7
Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο έναν φυσικό αριθμό `k` και θα καλεί την αναδρομική συνάρτηση `int find_sum(int k)`. Η συνάρτηση θα δέχεται ως όρισμα το `k` και θα επιστρέφει στη `main()` το άθροισμα $\sum_{i=1}^{k}(i+1)^2$, το οποίο και θα τυπώνεται στην οθόνη.

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Deitel, H. & Deitel, P. (2005), *Ασκήσεις - Προγράμματα σε C*, Εκδόσεις Γκιούρδα.

Horton, I. (2006), *Beginning C – from Novice to Professional*, 4th ed., Apress.

Prata, S. (2014), *C Primer Plus*, 6th ed., Addison-Wesley.

5. Πίνακες – αλφαριθμητικά

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στις έννοιες του πίνακα και του πίνακα χαρακτήρων (αλφαριθμητικού). Μελετώνται τόσο οι μονοδιάστατοι όσο και οι πολυδιάστατοι πίνακες σε ό,τι αφορά τα χαρακτηριστικά τους και τις λειτουργίες τους (δήλωση, ορισμός, ανάγνωση και εκτύπωση στοιχείων πίνακα, ανάθεση τιμής). Περιγράφεται σε αδρές γραμμές (χωρίς δείκτες) η χρήση πινάκων ως όρισμα συναρτήσεων. Αναλύονται τα αλφαριθμητικά και οι πίνακες αλφαριθμητικών και παρουσιάζονται οι βασικές συναρτήσεις διαχείρισης αλφαριθμητικών, καθώς και οι συναρτήσεις μετατροπής αλφαριθμητικών σε αριθμητικές τιμές.

Λέξεις κλειδιά
μονοδιάστατοι πίνακες, πολυδιάστατοι πίνακες, αλφαριθμητικά, πίνακες αλφαριθμητικών, αρχικοποίηση πινάκων, gets, puts, strlen, strcpy, strcat, strcmp, atoi, atol, atof, πίνακες μεταβλητού μήκους, καθοριστές.

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

5.1. Μονοδιάστατοι πίνακες

Ο πίνακας είναι μία *συλλογή δεδομένων του ίδιου τύπου*, τα οποία είναι αποθηκευμένα σε διαδοχικές θέσεις μνήμης.

Η δήλωση του πίνακα ακολουθεί τον εξής φορμαλισμό:

```
τύπος_δεδομένου όνομα_πίνακα[μέγεθος];
```

Διακρίνονται τρία τμήματα: *α)* ο τύπος δεδομένων (**float**, **int**, **char**, **double**), *β)* το όνομα του πίνακα και *γ)* ο αριθμός των στοιχείων του πίνακα. Έτσι, μία τυπική δήλωση ενός πίνακα 100 στοιχείων κινητής υποδιαστολής απλής ακρίβειας είναι η ακόλουθη:

```
float array[100];
```

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

array[0]: ***πρώτο*** στοιχείο του πίνακα
array[1]: ***δεύτερο*** στοιχείο του πίνακα
array[99]: ***τελευταίο*** (εκατοστό) στοιχείο του πίνακα

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

float array[5]={1.0,2.22,-4.2,6.7,338}: αρχικοποιούνται και τα 5 στοιχεία του πίνακα **array**.

float array[5]={1.0,2.22,-4.2}: αρχικοποιούνται τα 3 πρώτα στοιχεία του πίνακα **array**, δηλαδή τα **array[0]**, **array[1]**, **array[2]**.

Στο πρότυπο της γλώσσας C99 έχουν εισαχθεί οι **καθοριστές** (designators), οι οποίοι επιτρέπουν τη στοχευμένη αρχικοποίηση στοιχείων σε έναν πίνακα. Είναι ακέραιοι αριθμοί και λειτουργούν ως αριθμοδείκτες, καθώς γράφονται μέσα σε αγκύλες και καθορίζουν τη θέση στον πίνακα στην οποία θα τοποθετηθεί η αρχική τιμή. Η πρόταση

```
float array[5]={[2]=1.7, [4]=-19.3};
```

δημιουργεί τον πίνακα **array** και αρχικοποιεί το τρίτο και το πέμπτο στοιχείο του πίνακα με τις τιμές **1.7** και **-19.3**, αντίστοιχα.
Ένα επιπρόσθετο πλεονέκτημα των καθοριστών είναι ότι μπορούν να τοποθετηθούν με οποιαδήποτε σειρά κι όχι αποκλειστικά με αύξουσα, όπως φαίνεαι παρακάτω:

```
float array[9]={[2]=1.7, [7]=-19.3, [5]=13};
```

Εάν το μέγεθος ενός πίνακα έχει οριστεί με τη δήλωση, π.χ. **array[9]**, ο καθοριστές μπορούν να λάβουν τιμή από 0 έως και 8. Εάν όμως δεν οριστεί το μέγεθος του πίνακα, καθορίζεται έμμεσα από τη μέγιστη τιμή καθοριστή, ο οποίος δεν πρέπει να είναι αρνητικός ακέραιος. Στην ακόλουθη πρόταση

```
float array[]={[2]=1.7, [15]=-19.3, [5]=13};
```

το μέγεθος του πίνακα καθορίζεται στο **16** (**15+1**).

Η ανάγνωση και εκτύπωση ενός πίνακα γίνονται κατά στοιχείο, με τους κανόνες που ισχύουν για κάθε τύπο δεδομένου:

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

Παρατηρήσεις:
1. Όταν αποδίδονται αρχικές τιμές μπορεί να παραληφθεί το μέγεθος του πίνακα. Το μέγεθος του πίνακα καθορίζεται από τον αριθμό των αρχικών τιμών που δίδονται. Η παρακάτω δήλωση

```
char array[ ] = {'D','k','$','q'};
```

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

```
array[0]='D'
array[1]='k'
array[2]='$'
array[3]='q'
```

2. Το γεγονός ότι οι δείκτες των στοιχείων ενός πίνακα ξεκινούν από το **0** κι όχι από το **1** μπορεί να αποτελεί μία ιδιάζουσα ιδιότητα της γλώσσας C αλλά πηγάζει από τη φιλοσοφία της να αποτελεί μεν γλώσσα προγραμματισμού υψηλού επιπέδου αλλά ταυτόχρονα ο προγραμματισμός σε C να βρίσκεται κοντά στην αρχιτεκτονική του υπολογιστή. Εφόσον το **0** αποτελεί το σημείο εκκίνησης για τους υπολογιστές, εάν η αρίθμηση των στοιχείων πίνακα ξεκινούσε από το **1**, ο μεταγλωττιστής θα έπρεπε να αφαιρέσει τη μονάδα από κάθε αναφορά σε δείκτη στοιχείου (οι δείκτες θα παρουσιαστούν στο επόμενο κεφάλαιο), για να ληφθεί η διεύθυνση ενός στοιχείου.

3. Υπάρχει διαφορά ανάμεσα στη δήλωση πίνακα και στην αναφορά στοιχείου πίνακα. Σε μία δήλωση, ο αριθμός μέσα στις αγκύλες καθορίζει το μέγεθος του πίνακα. Σε μία αναφορά στοιχείου πίνακα, ο αριθμοδείκτης προσδιορίζει το στοιχείο του πίνακα, στο οποίο αναφερόμαστε. Π.χ. στη δήλωση **int array[100];** το **100** δηλώνει τον αριθμό των στοιχείων του πίνακα. Αντίθετα, στη **array[7]=167;** το **7** δηλώνει το 8^{ο} στοιχείο του πίνακα, στο οποίο αποδίδεται η τιμή **167**.

4. Μπορεί να βρεθεί το μέγεθος σε bytes ενός πίνακα χρησιμοποιώντας τον τελεστή **sizeof**. Για παράδειγμα, εάν θεωρηθεί ο πίνακας **int array[100];** η έκφραση **sizeof(array)** δίνει τιμή **400**, επειδή ο πίνακας αποτελείται από 100 ακεραίους των 4 bytes.

Στον **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(array[0])** δίνει τιμή **4**.

Χρησιμοποιώντας έναν συνδυασμό των παραπάνω μπορεί να βρεθεί ο αριθμός των στοιχείων του πίνακα. Η έκφραση **sizeof(array)/sizeof(array[0])** δίνει **100**, τον αριθμό δηλαδή των στοιχείων του πίνακα **array**.

5.1.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο διαδοχικά 6 ακέραιους αριθμούς του ΛΟΤΤΟ. Θα πρέπει να λαμβάνεται πρόνοια, ώστε, όταν ένας αριθμός που δίνεται από το πληκτρολόγιο είτε δεν ανήκει στο **[1, 49]** είτε έχει δοθεί 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

int main()   {
  int lotto[6],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++) /* Να μην επαναληφθεί προηγούμενος αριθμός */
          if (lotto[j]==lotto[i]) deikt++;
       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] );

  return 0;
}
```

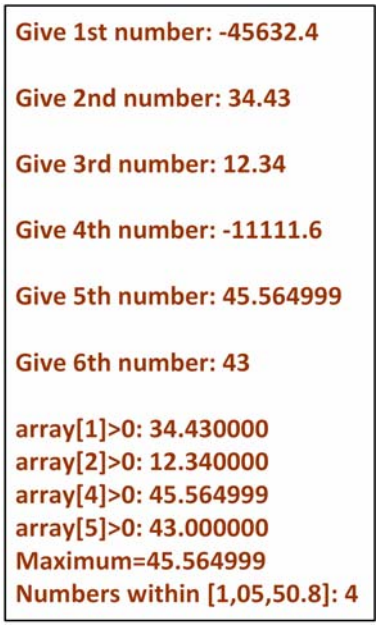
```
Number 1, enter your choice:45

Number 2, enter your choice:67

Invalid choice!! Try again

Number 2, enter your choice:-1

Invalid choice!! Try again

Number 2, enter your choice:2

Number 3, enter your choice:2

Invalid choice!! Try again

Number 3, enter your choice:5

Number 4, enter your choice:7

Number 5, enter your choice:8

Number 6, enter your choice:9

Number 1 is set to 45
Number 2 is set to 2
Number 3 is set to 5
Number 4 is set to 7
Number 5 is set to 8
Number 6 is set to 9
```

Εικόνα 5.1 *Η έξοδος του προγράμματος του παραδείγματος 5.1.1*

5.1.2. Παράδειγμα

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

```
#include <stdio.h>
#include <math.h>  /* για τη συνάρτηση fabs() */

#define N 6
#define lower 1.05
#define upper 50.8

int main()
{
  float arrayay[N],maxim;
  int i,count=0;
  for (i=0;i<N;i++)
  {
    switch(i) /* για να εμφανιστεί σωστά η αρίθμηση */
    {
      case 0:
        printf( "\nGive 1st number: " );
      break;
      case 1:
        printf( "\nGive 2nd number: " );
      break;
      case 2:
        printf( "\nGive 3rd number: " );
      break;
      default:
        printf( "\nGive %dth number: ",i+1 );
      break;
    }
    scanf( "%f",&arrayay[i] );
  }
  maxim=arrayay[0];
  printf( "\n" );
  for  (i=0;i<N;i++)       /* i=0 για να μπουν όλοι οι έλεγχοι που απαιτούν τα σκέλη (α), (β), (γ) σε ένα βρόχο. Εάν ο βρόχος αφορούσε μόνο την εξαγωγή του μεγίστου, ο μετρητής θα ξεκινούσε από το 1 */
  {
    if (arrayay[i]==fabs(arrayay[i]))
      printf( "arrayay[%d]>0: %f\n",i,arrayay[i] );
    if (arrayay[i]>maxim) maxim=arrayay[i];
    if ((arrayay[i]>=lower) && (arrayay[i]<=upper)) count++;
  }
  printf( "Maximum=%f\n",maxim );
  printf( "Numbers within [%d,%d]: %d\n",lower,upper,count );

  return 0;
}
```



Εικόνα 5.2 *Η έξοδος του προγράμματος του παραδείγματος 5.1.2*

5.1.3. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα δέχεται από το πληκτρολόγιο έναν 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 $y = y_{n-1}y_{n-2}...y_1y_0 = x_0x_1...x_{n-2}x_{n-1}$.

```
#include <stdio.h>
#define N 7

int main()
{
  int x,y,array[N],z[N],i;
  printf( "Give 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—)
  {
    array[i]=y/z[i];
    y=y%z[i];
  }
  array[0]=y;
  y=array[N-1] ;
  for (i=1;i<N;i++) y=y+z[i]*array[N-(i+1)];
  printf( "\nx=%d, y=%d\n",x,y );

  return 0 ;
}
```

Εικόνα 5.3 *Η έξοδος του προγράμματος του παραδείγματος 5.1.3*

5.2. 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 γλώσσα C δεν θέτει περιορισμό στον αριθμό των διαστάσεων των πινάκων.

Ο πολυδιάστατος πίνακας **N** διαστάσεων αποθηκεύεται στη μνήμη ως μία ακολουθία στοιχείων μίας διάστασης, ωστόσο μπορεί να θεωρηθεί ως ένας μονοδιάστατος πίνακας, όπου κάθε στοιχείο του είναι ένας πίνακας **N-1** διαστάσεων, δηλαδή ένας πολυδιάστατος πίνακας είναι ένας *πίνακας πινάκων*. Για παράδειγμα, έστω ο επόμενος τριδιαγώνιος πίνακας, οποίος εμφανίζει μη μηδενικά στοιχεία μόνο στην κύρια διαγώνιο και τις δύο διαγωνίους εκατέρωθέν της:

17	62	0	0	0
25	28	9	0	0
0	6	33	28	0
0	0	123	49	3
0	0	0	76	13

Εικόνα 5.4 *Τριδιαγώνιος πίνακας.*

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

```
int tridiag[5][5]=  {  {17, 62, 0, 0, 0},
                       {25, 28, 9, 0, 0},
                       {0, 6, 33, 28, 0},
                       {0, 0, 123, 49, 3},
                       {0, 0, 0, 76, 13}
                    };
```

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 καθοριστούν τόσοι δείκτες όσοι είναι αναγκαίοι. Έτσι, η έκφραση

```
tridiag[1]
```

αναφέρεται στη δεύτερη γραμμή του πίνακα, ενώ η έκφραση

```
tridiag[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 array[2][3]=  {  {0, 1, 2},
                     {3, 4, 5}
                  };
```

αποθηκεύεται όπως φαίνεται στο **Σχήμα 5.1**:

	διεύθυνση	τιμή
array[0][0]	1000	0
array[0][1]	1004	1
array[0][2]	1008	2
array[1][0]	1012	3
array[1][1]	1016	4
array[1][2]	1020	5

Σχήμα 5.1 *Αποθήκευση πολυδιάστατου (δισδιάστατου) πίνακα*

Όταν δηλώνεται ένας πίνακας χωρίς να αρχικοποιηθεί, το περιεχόμενο των θέσεών του είναι απροσδιόριστο. Κατά συνέπεια, δεν πρέπει ποτέ να χρησιμοποιείται ένα στοιχείο πίνακα, εάν δεν έχει λάβει πρώτα τιμή. Για παράδειγμα, ο ακόλουθος κώδικας

```
int i,j,array[3][3];
for (i=0;i<3;i++)
     for (j=0;j<3;j++) printf( "arr[%d][%d]=%d\n",i,j,array[i][j] );
```

δίνει τα ακόλουθα αποτελέσματα, που παρουσιάζονται στην **Εικόνα 5.5**, τα οποία προφανώς είναι μη αναμενόμενα και μπορούν να οδηγήσουν σε ανεπιθύμητες καταστάσεις:

```
array[0][0]=1
array[0][1]=0
array[0][2]=4200201
array[1][0]=0
array[1][1]=0
array[1][2]=0
array[2][0]=3
array[2][1]=0
array[2][2]=21
```

Εικόνα 5.5 *Η έξοδος του προγράμματος*

5.2.1. Αρχικοποίηση πολυδιάστατων πινάκων

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 array[4][4]= {  {1, 2, 3, 4},
                    {5, 6},
                    {7, 8, 9}
                 };
```

η μεταβλητή **array** δηλώνεται ως πίνακας 4 γραμμών και 4 στηλών. Ωστόσο, έχουν αποδοθεί τιμές μόνο για τις 3 πρώτες γραμμές του πίνακα και μάλιστα για τη δεύτερη γραμμή έχει οριστεί η τιμή μόνο των δύο πρώτων στοιχείων, ενώ για την τρίτη γραμμή δεν έχει οριστεί η τιμή του τελευταίου στοιχείου της. Η ανωτέρω δήλωση οδηγεί στον ακόλουθο πίνακα:

1	**2**	**3**	**4**
5	**6**	**0**	**0**
7	**8**	**9**	**0**
0	**0**	**0**	**0**

Εάν δεν συμπεριληφθούν τα ενδιάμεσα άγκιστρα:

```
int array[4][4]= {  1, 2, 3, 4
                    5, 6
                    7, 8, 9 };
```

τα στοιχεία θα αποθηκευτούν στις διαδοχικές θέσεις μνήμης που έχουν δεσμευτεί για τον πίνακα **array** και θα προκύψει ο ακόλουθος πίνακας:

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**0**	**0**	**0**
0	**0**	**0**	**0**

Παρατηρήσεις:

1. Όπως και με τους πίνακες μίας διάστασης, έτσι και στους πολυδιάστατους πίνακες, εάν δεν δοθεί το μέγεθος (οι διαστάσεις) του πίνακα, ο μεταγλωττιστής θα το καθορίσει αυτόματα με βάση τον αριθμό αρχικών τιμών που παρουσιάζονται. 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 array[ ][4][2]= {  {  {1, 2}, {3, 4}, {5, 6}, {7, 8} },
                       {  {9, 10}, {11, 12}, {13, 14}, {15, 16}  }
                    };
```

Με την ανωτέρω δήλωση ο **array** δηλώνεται αυτόματα ως πίνακας **2x4x2**.

2. Η δήλωση

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

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

3. Οι καθοριστές εφαρμόζονται και στους πολυδιάστατους πίνακες. Η ακόλουθη πρόταση

```
float identityMatrix[2][2]={[0][0]=1.0, [1][1]=1.0};
```

δημιουργεί έναν μοναδιαίο πίνακα διαστάσεων **2x2**. Ως συνήθως, η τιμή των μη αρχικοποιούμενων στοιχείων τίθεται αυτόματα στο **0**.

4. Η πρόταση

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

είναι λανθασμένη στη γλώσσα C, αλλά δεν εντοπίζεται από τον μεταγλωττιστή και οδηγεί σε μη αναμενόμενα – επομένως ανεπιθύμητα – αποτελέσματα. Η σωστή πρόταση είναι

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

5.2.2. Παράδειγμα

Σε μία εταιρεία εργάζονται 5 πωλητές. Οι συνολικές μηνιαίες αποδοχές κάθε πωλητή προκύπτουν από το άθροισμα του βασικού μισθού (700 €) και του 9% επί του μηνιαίου τζίρου που πέτυχε ο πωλητής τον εκάστοτε μήνα. Π.χ. εάν ο μηνιαίος τζίρος που πέτυχε ένας πωλητής είναι 12000€, τότε οι αποδοχές του αυτόν τον μήνα θα είναι $700 + 0.09 \cdot 12000 = 1780$ €.

Με βάση τα παραπάνω να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- Θα δημιουργεί πίνακα αριθμών κινητής υποδιαστολής, διαστάσεων **5x3**.
- Με χρήση κατάλληλης επαναληπτικής πρότασης θα δίνεται από το πληκτρολόγιο ο μηνιαίος τζίρος κάθε πωλητή και ακολούθως θα υπολογίζονται οι μηνιαίες αποδοχές, οι οποίες θα αποθηκεύονται στον ανωτέρω πίνακα και θα εμφανίζονται στην οθόνη. Ταυτόχρονα θα υπολογίζεται ο συνολικός τζίρος και θα εμφανίζεται στην οθόνη.
- Ο τζίρος κάθε πωλητή και το ποσοστό που συνεισφέρει κάθε πωλητής στον συνολικό τζίρο θα αποθηκεύονται στον ανωτέρω πίνακα. Τα ποσοστά θα εμφανίζονται στην οθόνη.
- Θα υπολογίζεται και θα εμφανίζεται στην οθόνη ο αριθμός των πωλητών, για τους οποίους οι μηνιαίες αποδοχές εντάσσονται σε κάποια από τις ακόλουθες κατηγορίες.

I. 700-1500€
II. 1501-2000€
III. περισσότερα από 2000€

```
#include <stdio.h>
#define N 5

#define BS 700
#define COEF 0.09
#define FC 1500
#define SC 1500

int main()
{
  float salesman[N][3],gross=0.0;
  int i,cat[3]={0,0,0};

  for (i=0;i<N;i++)
  {
    printf( "\nSalesman no %d, give the gross sales:  ",i+1 );
    scanf( "%.2f",&salesman[i][1] );
    salesman[i][0]=BS+COEF*salesman[i][1];
    gross=gross+salesman[i][1];
  }
  printf( "\nTotal gross sales: %.2f",gross );

  for (i=0;i<N;i++)
  {
    printf( "\nSalesman no %d, salary: %.2f",i+1,salesman[i][0] );
    salesman[i][2]=100*salesman[i][1]/gross;
    printf(" \nSalesman no %d contribution to the total gross sales:
%.2f\n",i+1,salesman[i][2] );
    if (salesman[i][0]>2000) cat[2]++;
    else if (salesman[i][0]>1500) cat[1]++;
    else cat[0]++;
  }

  for (i=0;i<3;i++)
```

```
    {
      if (cat[i]==0)
         printf( "\nNo salesman falls into category %d\n",i+1 );
      else if (cat[i]==1)
         printf( "\n1 salesman falls into category %d\n",i+1 );
      else printf( "\n%d salesmen fall into category %d\n",cat[i],i+1 );
    }

    return 0;
}
```

```
Salesman no 1, give the gross sales: 12000

Salesman no 2, give the gross sales: 15000

Salesman no 3, give the gross sales: 17000

Salesman no 4, give the gross sales: 30000

Salesman no 5, give the gross sales: 25000

Total gross sales: 99000.00
Salesman no 1, salary: 1780.00
Salesman no 1 contribution to the total gross sales: 12.12

Salesman no 2, salary: 2050.00
Salesman no 2 contribution to the total gross sales: 15.15

Salesman no 3, salary: 2230.00
Salesman no 3 contribution to the total gross sales: 17.17

Salesman no 4, salary: 3400.00
Salesman no 4 contribution to the total gross sales: 30.30

Salesman no 5, salary: 2950.00
Salesman no 4 contribution to the total gross sales: 25.25

No salesman falls into category 1

1 salesman falls into category 2

4 salesmen fall into category 3
```

Εικόνα 5.6 *Η έξοδος του προγράμματος του παραδείγματος 5.2.2*

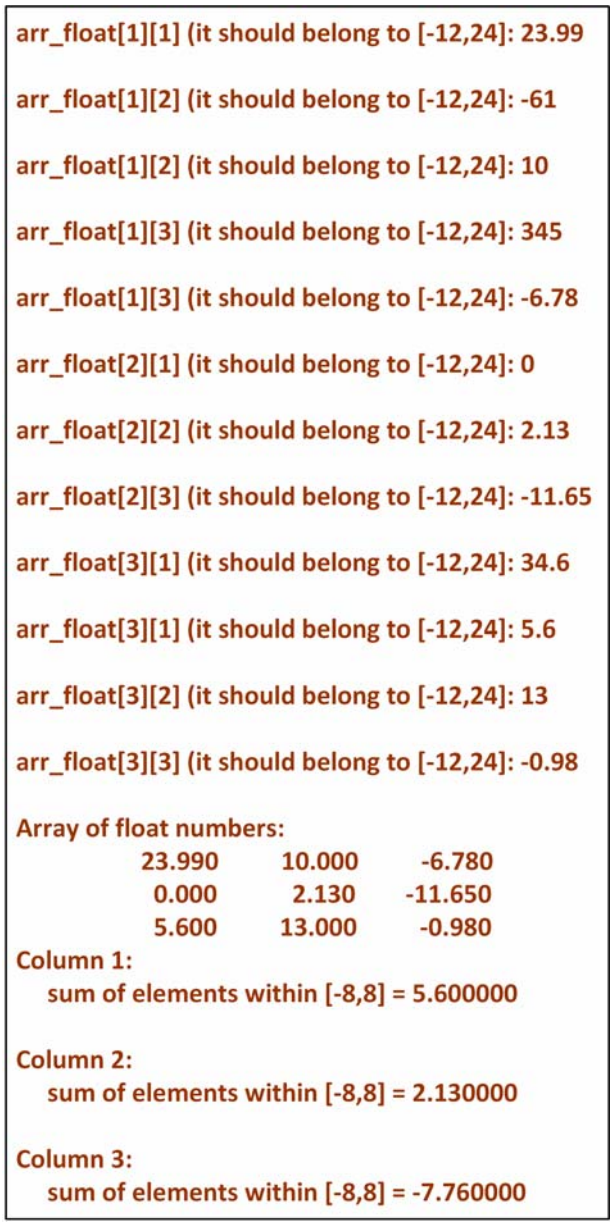
5.2.3. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- Θα λαμβάνει από το πληκτρολόγιο τις τιμές ενός πίνακα αριθμών κινητής υποδιαστολής `arrFloat[3][3]`. Μέσω κατάλληλης επαναληπτικής πρότασης, οι τιμές θα πρέπει υποχρεωτικά να κείνται στο διάστημα **[-12, 24]**. Ο προκύπτων πίνακας θα εμφανίζεται στην οθόνη.

- Για κάθε στήλη του πίνακα **arrFloat** θα υπολογίζει το άθροισμα των στοιχείων της στήλης, οι τιμές των οποίων ανήκουν στο διάστημα **[-8, 8]**, και θα εμφανίζει το άθροισμα στην οθόνη.

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

#define N 3

#define LOWER_1 -12
#define UPPER_1 24

#define LOWER_2 -8
#define UPPER_2 8

int main()
{
  int i,j,ln[N],count_1,count_2;
  float arrFloat[N][N],columnSum;
  char arrChar[N][26];

  for (i=0;i<N;i++)
     for (j=0;j<N;j++)
     {
       do
       {
         printf( "\narr_float[%d][%d] (it should belong to [%d,%d]):
",i+1,j+1,LOWER_1,UPPER_1 );
         scanf("%f",&arrFloat[i][j]);
       } while ((arrFloat[i][j]<LOWER_1) ||
              (arrFloat[i][j]>UPPER_1));
     }

     printf( "\n\nArray of float numbers:" );
     for (i=0;i<N;i++)
     {
       printf("\n");
       for (j=0;j<N;j++)  printf( "\t%10.3f",arrFloat[i][j] );
     }

   for (j=0;j<N;j++)
   {
     columnSum=0;
     for (i=0;i<N;i++)
     {
       if ((arrFloat[i][j]>LOWER_2) && (arrFloat[i][j]<=UPPER_2))
         columnSum+=arrFloat[i][j];
     }
     printf( "\nColumn %d:\n",j+1 );
     printf(     "\tsum     of     elements     within     [%d,%d]     =
%f\n",LOWER_2,UPPER_2,columnSum );
   }

   return 0;
}
```



Εικόνα 5.7 *Η έξοδος του προγράμματος του παραδείγματος 5.2.3*

5.3. Πίνακες μεταβλητού μήκους

Έως τώρα η δήλωση ενός πίνακα προϋπέθετε ότι οι διαστάσεις του θα ήταν εκ των προτέρων γνωστές, δηλαδή κατά τον χρόνο μεταγλώττισης του προγράμματος. Αυτός είναι ένας σημαντικός περιορισμός, καθώς σε πλήθος εφαρμογών οι διαστάσεις ενός πίνακα *προκύπτουν* κατά την εκτέλεση του προγράμματος (π.χ. ως αποτέλεσμα της εκτέλεσης μίας πρότασης **`if`**–**`else`**) .

Στο πρότυπο C99 της γλώσσας C οι πίνακες ενισχύθηκαν με τη δυνατότητα να δηλώνονται οι διαστάσεις τους κατά τον χρόνο εκτέλεσης του προγράμματος. Οι πίνακες αυτοί ονομάζονται «πίνακες μεταβλητού μήκους» (variable length arrays, VLA). Σημειώνεται ότι οι διαστάσεις τους καθορίζονται άπαξ, όπως στους κλασικούς πίνακες, και δεν μπορούν να μεταβληθούν. Μία τέτοια δυνατότητα δίνει η δυναμική διαχείριση μνήμης, που θα παρουσιαστεί στο 7^{ο} Κεφάλαιο.

5.3.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα χρησιμοποιήσει πίνακες μεταβλητού μήκους, για να επιτελέσει τα ακόλουθα:

- Θα δέχεται από το πληκτρολόγιο τις διαστάσεις δύο πινάκων **a** και **b**.
- Θα δημιουργεί τους δύο πίνακες κατά τον χρόνο εκτέλεσης του προγράμματος.
- Θα τροφοδοτεί τους πίνακες με δεδομένα και θα τους εμφανίζει στην οθόνη.
- Θα υλοποιεί τις πράξεις της πρόσθεσης, αφαίρεσης και πολλαπλασιασμού πινάκων, Προς τούτο θα υπάρχει μενού επιλογής πράξης. Μετά την επιλογή θα προηγείται έλεγχος των διαστάσεων των πινάκων και θα ακολουθεί η εκτέλεση των υπολογισμών και η εμφάνιση του προκύπτοντος πίνακα στην οθόνη.

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

#define ADD 1
#define SUB 2
#define MUL 3

int main()
{
  int i,j,k,a_ln,a_cl,b_ln,b_cl,c_ln,c_cl,d_ln,d_cl;
  float counter;

  /* Ανάγνωση των δεδομένων: Χάριν ευκολίας, τα δεδομένα
   δεν αναγιγνώσκονται αλλά δίνονται από έναν αλγόριθμο  */
  printf( "\nGive the number of lines for array a: " );
  scanf( "%d",&a_ln );
  printf( "\nGive the number of columns for array a: " );
  scanf( "%d",&a_cl );
  printf( "\nGive the number of lines for array b: " );
  scanf("%d",&b_ln);
  printf( "\nGive the number of columns for array b: " );
  scanf( "%d",&b_cl );
  float a[a_ln][a_cl],b[b_ln][b_cl]; /* δημιουργία πινάκων VLA */
  for (i=0;i<a_ln;i++) {
     for (j=0;j<a_cl;j++) {
       a[i][j]=2.0*(i+2)*(j+4)/((i+1)*(i+1));
     }
  }
    for (i=0;i<b_ln;i++)  {
    for (j=0;j<b_cl;j++)    {
        b[i][j]=4.0*(i+2)*(j+1)/((i+1)*(i+1));
     }
  }
  /* Εκτύπωση των πινάκων  */
  printf( "\n\n\nArray a:\n\t" );
  for (i=0;i<a_ln;i++) {
    for (j=0;j<a_cl;j++) {
      printf( "%.3f\t",a[i][j] );
    }
    printf( "\n\t" );
  }
  printf( "\n\n\nArray b:\n\t" );
```

```
for (i=0;i<b_ln;i++) {
  for (j=0;j<b_cl;j++) {
    printf("%.3f\t",b[i][j]);
  }
  printf("\n\t");
}
int choice;
do
{
  printf( "\n Select one of the following:" );
  printf( "\n\t\t\t %d -> + (addition)\n",ADD );
  printf( "\n\t\t\t %d -> - (subtraction)\n",SUB );
  printf( "\n\t\t\t %d -> * (multiplication)\n",MUL );
  scanf( "%d",&choice );
} while ((choice!=1) && (choice!=2) && (choice!=3));
switch(choice)
{
  case ADD:
    if ((a_ln==b_ln) && (a_cl==b_cl))
    {
      printf( "\n\n\nArray a+b:\n\t" );
      for (i=0;i<a_ln;i++)
      {
        for (j=0;j<a_cl;j++)
        {
          printf(" %.3f\t",a[i][j]+b[i][j] );
        }
        printf(" \n\t" );
      }
    }
    else
      printf( "\nERROR!! Arrays' dimensions are inconsistent!\n" );
  break;
  case SUB:
    if ((a_ln==b_ln) && (a_cl==b_cl))
    {
      printf( "\n\n\nArray a-b:\n\t" );
      for (i=0;i<a_ln;i++)
      {
        for (j=0;j<a_cl;j++)
        {
          printf( "%.3f\t",a[i][j]-b[i][j] );
        }
        printf( "\n\t" );
      }
    }
    else
      printf( "\nERROR!! Arrays' dimensions are inconsistent!\n" );
  break;
  default:
    if (a_cl==b_ln)
    {
      printf( "\n\n\nArray axb:\n  " );
      for (i=0;i<a_ln;i++)
      {
```

```
            for (j=0;j<b_cl;j++)
            {
              counter=0.0;
              for (k=0;k<a_cl;k++) counter=counter+(a[i][k]*b[k][j]);
              printf( "%8.3f  ",counter );
            }
            printf("\n  ");
          }
        }
        else
         printf( "\nERROR!! Arrays' dimensions are inconsistent!\n" );
      break;
  }   /* τέλος της switch */

  return 0;
}
```

```
Give the number of lines for array a: 3

Give the number of columns for array a: 4

Give the number of lines for array b: 4

Give the number of columns for array b: 3


Array a:
        16.000  20.000  24.000  28.000
        6.000   7.500   9.000   10.500
        3.556   4.444   5.333   6.222


Array b:
        8.000  16.000  24.000
        3.000  6.000   9.000
        1.778  2.500   5.333
        1.250  2.500   3.750

Select one of the following:
                        1 -> + (addition)
                        2 -> - (subtraction)
                        3 -> * (multiplication)
3


Array axb:
   265.667  531.333  797.000
    99.625  199.250  298.875
    59.037  118.074  177.111
```

Εικόνα 5.8 *Η έξοδος του προγράμματος του παραδείγματος 5.3.1*

5.4. Πίνακες ως παράμετροι συναρτήσεων

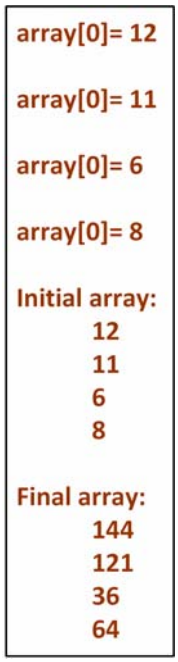
Εάν κατά την κλήση μίας συνάρτησης η πραγματική παράμετρος είναι όνομα πίνακα (πχ. **arr**), δεν αποστέλλεται στη συνάρτηση ολόκληρος ο πίνακας αλλά μόνο η διεύθυνση μνήμης του πρώτου byte του πρώτου στοιχείου του πίνακα. Η παράμετρος στη δήλωση της συνάρτησης είναι ένα όνομα τοπικού πίνακα (π.χ. **localArr**), ο οποίος κρατά ένα αντίγραφο της ίδιας διεύθυνσης. Με τον τρόπο αυτό η μεταβλητή **localArr** στην πραγματικότητα διαχειρίζεται τις θέσεις μνήμης που καταλαμβάνει ο πίνακας της καλούσας συνάρτησης, κατά συνέπεια μπορεί να μεταβάλλει με έμμεσο τρόπο τις τιμές του πίνακα και αυτές οι μεταβολές να διατηρηθούν και μετά την ολοκλήρωση της κλήσης της συνάρτησης.

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 στον μεταγλωττιστή η διεύθυνση του πρώτου byte και η απόσταση (αριθμός bytes) μεταξύ των στοιχείων. Ο μηχανισμός κλήσης συνάρτησης με όρισμα πίνακα στηρίζεται στην έννοια των δεικτών και θα μελετηθεί εκτενώς στο επόμενο κεφάλαιο.

5.4.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα λαμβάνει 4 ακέραιους από το πληκτρολόγιο και θα τους αποδίδει σε πίνακα ακεραίων (**array[4]**). Στη συνέχεια, θα καλείται η συνάρτηση **void pwr(int localArray[], int arraySize)**, 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 εμφάνιση των νέων τιμών του **array** στην οθόνη.

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

void pwr (int localArray[], int arraySize);

int main()
{
  int array[4],i;
  for (i=0;i<4;i++)
  {
       printf( "\narray[%d]= ",i );
       scanf( "%d",&array[i] );
  }

  printf( "\n\nInitial array:\n" );
  for (i=0;i<4;i++) printf( "\t%d\n",array[i] );

  pwr(array,4);

  printf( "\n\nFinal array:\n" );
  for (i=0;i<4;i++) printf( "\t%d\n",array[i] );

  return 0;
}

void pwr (int localArray[], int arraySize)
{
   int i;
   for (i=0;i<4;i++) localArray[i]=localArray[i]*localArray[i];
}
```



Εικόνα 5.9 *Η έξοδος του προγράμματος του παραδείγματος 5.4.1*

5.5. Το αλφαριθμητικό

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

Η δήλωση του αλφαριθμητικού ακολουθεί τον εξής φορμαλισμό:

```
char όνομα_αλφαριθμητικού[μήκος_αλφαριθμητικού];
```

Η δήλωση περιλαμβάνει τρία τμήματα: *α)* τον τύπο δεδομένου, ο οποίος είναι πάντοτε char, *β)* το όνομα του αλφαριθμητικού και *γ)* το μήκος του, το οποίο πάντοτε περιλαμβάνει μία θέση παραπάνω από το μέγιστο σε μήκος προσδοκώμενο αλφαριθμητικό, καθώς πρέπει να λαμβάνεται πρόνοια για την αποθήκευση του μηδενικού χαρακτήρα. Έτσι, μία τυπική δήλωση ενός αλφαριθμητικού 100 χαρακτήρων έχει την ακόλουθη μορφή:

```
char stringName[101];
```

Τα αλφαριθμητικά μπορούν να εμφανίζονται μέσα στον κώδικα όπως οι αριθμητικές σταθερές, αποτελώντας τις **αλφαριθμητικές σταθερές**. Αλφαριθμητικές σταθερές χρησιμοποιήθηκαν στο Κεφάλαιο 2, περικλειόμενες σε διπλά εισαγωγικά. Για την αποθήκευσή τους χρησιμοποιούνται πίνακες χαρακτήρων, με τον μεταγλωττιστή να θέτει αυτόματα στο τέλος των αλφαριθμητικών έναν μηδενικό χαρακτήρα, για να προσδιορίσει το τέλος του. Έτσι, η αλφαριθμητική σταθερά `"Constant string"` απαιτεί για αποθήκευση **16** bytes, όπως φαίνεται παρακάτω:

```
'C','o','n','s','t','a','n','t',' ','s','t','r','i','n','g','\0'
```

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

Ένας **πίνακας αλφαριθμητικών** είναι ένας δισδιάστατος πίνακας, κάθε γραμμή του οποίου φιλοξενεί ένα αλφαριθμητικό. Μία τυπική δήλωση ενός πίνακα 10 αλφαριθμητικών των 50 χαρακτήρων έχει την ακόλουθη μορφή:

```
char stringArray[10][51];
```

5.6. Αρχικοποίηση αλφαριθμητικού

Η αρχικοποίηση αλφαριθμητικού μπορεί να γίνει με δύο τρόπους:

1. Με τη δήλωση, όπου ακολουθείται ο γενικός κανόνας απόδοσης αρχικής τιμής σε πίνακα:

```
char stringName[ ] = {'m','y',' ','s','t','r','i','n','g','\0'};
```

2. Με χρήση αλφαριθμητικής σταθεράς (προσφιλέστερος τρόπος):

```
char stringName[ ] = {"my string"};
```

Θα πρέπει να σημειωθεί ότι στη χρήση του γενικού κανόνα απόδοσης αρχικής τιμής σε πίνακα, ο προγραμματιστής πρέπει να περιλάβει ως τελευταία τιμή το **null**. Στον δεύτερο τρόπο απόδοσης αρχικής τιμής, αυτό το έργο το εκτελεί αυτόματα ο μεταγλωττιστής.

5.7. Είσοδος – έξοδος αλφαριθμητικών

5.7.1. Ανάγνωση αλφαριθμητικού

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

```
scanf( "%s", stringName );
```

διαβάζει από την κύρια είσοδο ένα αλφαριθμητικό και το αποθηκεύει στη μεταβλητή **stringName**. Δεν 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()**, το πρωτότυπο της οποίας βρίσκεται στο αρχείο κεφαλίδας **stdio.gr** και έχει τη γενική μορφή

```
gets(όνομα_αλφαριθμητικού)
```

Καλείται η **gets()** με το όνομα του πίνακα χαρακτήρων ως όρισμα. Η **gets()** αναθέτει το αλφαριθμητικό στον πίνακα χαρακτήρων **όνομα_αλφαριθμητικού**. Η **gets()** θα διαβάζει χαρακτήρες από το πληκτρολόγιο, έως ότου πατηθεί το **ENTER**.

Για να διαβαστεί ένα στοιχείο πίνακα αλφαριθμητικών, χρησιμοποιούνται οι προτάσεις

```
scanf( "%s", stringArray[i] );
```
και
```
gets(stringArray[i]);
```

Παρατηρήσεις:

1. Η συνάρτηση **scanf()** διαβάζει εσφαλμένα τα αλφαριθμητικά που περιέχουν λευκούς χαρακτήρες, καθώς σταματά την ανάγνωση στην εμφάνιση του πρώτου λευκού χαρακτήρα. Αντίθετα, η συνάρτηση **gets()** διαβάζει τους λευκούς χαρακτήρες.

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 stringName[100]** και το αλφαριθμητικό είναι μεγαλύτερο από το μέγεθος του **stringName**, ο πίνακας θα ξεπεραστεί. Επαφίεται στον προγραμματιστή να φροντίζει, ώστε να μην προκληθεί υπέρβαση ορίων.

3. Όταν χρησιμοποιούνται πίνακες αλφαριθμητικών, η πρόταση

```
scanf( "%c", stringArray[i][j] );
```

διαβάζει τον χαρακτήρα του αλφαριθμητικού **i+1**, ο οποίος βρίσκεται στη θέση **j+1**.

5.7.2. Εκτύπωση αλφαριθμητικού

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

```
printf( "My string" );
```

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

```
printf( "This is %s!!", stringName );
```

θα έχει ως αποτέλεσμα να τυπωθεί στην οθόνη η πρόταση

```
This is my string!!
```

Εναλλακτικά, η εκτύπωση αλφαριθμητικής σταθεράς και αλφαριθμητικού μπορεί να γίνει με χρήση της συνάρτησης **puts()**, το πρωτότυπο της οποίας βρίσκεται στο αρχείο κεφαλίδας **stdio.gr** και έχει τη γενική μορφή

```
puts(όνομα_αλφαριθμητικού)
```

Καλείται η **puts()** με το όνομα του πίνακα χαρακτήρων ως όρισμα, χωρίς δείκτη, π.χ. **puts(stringName)**. Σε αντιδιαστολή με τη συνάρτηση **printf()**, η **puts()** δεν παρέχει δυνατότητες μορφοποίησης της εξόδου.

Για να εκτυπωθεί ένα στοιχείο πίνακα αλφαριθμητικών, χρησιμοποιούνται οι προτάσεις

```
printf( "%s", stringArray[i] );
```
και
```
puts(stringArray[i]);
```

ενώ η πρόταση

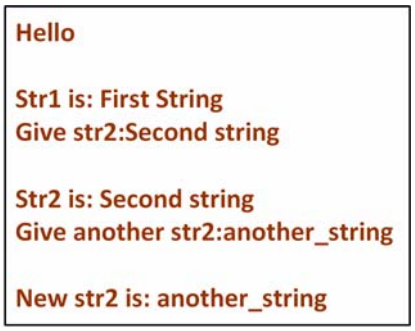
```
printf( "%c", stringArray[i][j] );
```

εκτυπώνει τον χαρακτήρα του αλφαριθμητικού **i+1**, ο οποίος βρίσκεται στη θέση **j+1**.

5.7.3.1. Παράδειγμα

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

```
#include <stdio.h>

#define STA "Hello"

int 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);

     return 0;
}
```



Εικόνα 5.10 *Η έξοδος του προγράμματος του παραδείγματος 5.7.3.1*

5.8. Μετατροπές αλφαριθμητικών σε αριθμητικές τιμές

Ένα αλφαριθμητικό που αποτελείται από ψηφία μπορεί να μετατραπεί σε αριθμό με χρήση των ακόλουθων συναρτήσεων, τα πρότυπα των οποίων βρίσκονται στο αρχείο κεφαλίδας **stdlib.h**:

- Η συνάρτηση **atoi()** δέχεται ως όρισμα ένα αλφαριθμητικό και επιστρέφει την ακέραια τιμή του ή, εφόσον δεν είναι εφικτή η μετατροπή, το μηδέν.
- Η συνάρτηση **atol()** δέχεται ως όρισμα ένα αλφαριθμητικό και επιστρέφει την τιμή του ως **long int** ή, εφόσον δεν είναι εφικτή η μετατροπή, το μηδέν.
- Η συνάρτηση **atof()** δέχεται ως όρισμα ένα αλφαριθμητικό και επιστρέφει την τιμή του ως αριθμό κινητής υποδιαστολής μονής ακρίβειας ή, εφόσον δεν είναι εφικτή η μετατροπή, το μηδέν.

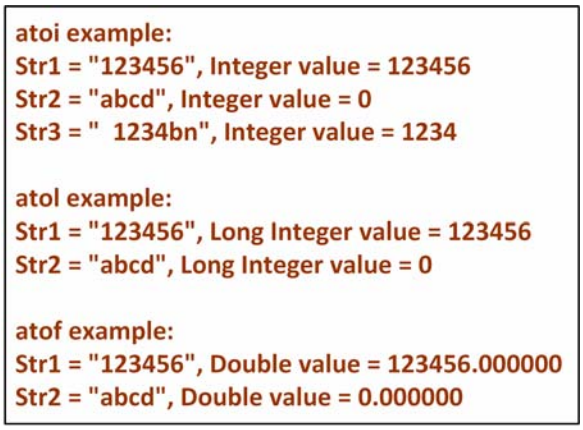
Το αλφαριθμητικό μπορεί να περιέχει κενά στην αρχή και το τέλος του. Η μετατροπή σταματά με την εμφάνιση του πρώτου μη αποδεκτού χαρακτήρα.

5.8.1. Παράδειγμα

Στο πρόγραμμα που ακολουθεί παρουσιάζονται τα χαρακτηριστικά των συναρτήσεων **atoi()**, **atol()**, **atof()**:

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

int main()
{
  int val1;
  long int val2;
  double val3;

  char str1[20]={"123456"},str2[20]={"abcd"},str3[20]={"  1234bn"};
  printf( "atoi example:\n" );
  val1 = atoi(str1);
  printf( "Str1 = \"%s\", Integer value = %d\n", str1, val1 );
  val1 = atol(str2);
  printf( "Str2 = \"%s\", Integer value = %d\n", str2, val1 );
  val1 = atol(str3);
  printf( "Str3 = \"%s\", Integer value = %d\n", str3, val1 );

  printf("\n\natol example:\n");
  val2 = atol(str1);
```

```
    printf( "Str1 = \"%s\", Long integer value = %ld\n", str1, val2 );
    val2 = atol(str2);
    printf( "Str2 = \"%s\", Long integer value = %ld\n", str2, val2 );

    printf( "\n\natof example:\n" );
    val3 = atof(str1);
    printf( "Str1 = \"%s\", Double value = %f\n", str1, val3 );
    val3 = atof(str2);
    printf( "Str2 = \"%s\", Double value = %f\n", str2, val3 );

    return(0);
}
```



Εικόνα 5.11 *Η έξοδος του προγράμματος του παραδείγματος 5.8.1*

5.9. Συναρτήσεις αλφαριθμητικών

Η γλώσσα C υποστηρίζει μία σειρά συναρτήσεων για τον χειρισμό των αλφαριθμητικών. Οι συναρτήσεις αυτές βρίσκονται στο αρχείο κεφαλίδας `string.h`. Οι πλέον συνήθεις παρουσιάζονται στον **Πίνακα 5.1**, 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 αλφαριθμητικού	`strlen()`	
Αντιγραφή αλφαριθμητικού	`stcpy()`	`strncpy()`
Συνένωση δύο αλφαριθμητικών	`strcat()`	`strncat()`
Σύγκριση δύο αλφαριθμητικών	`strcmp()`	`strncmp()`
Εύρεση χαρακτήρα σε αλφαριθμητικό	`strchr()`	`strrchr()`
Εύρεση αλφαριθμητικού σε αλφαριθμητικό	`strstr()`	

Πίνακας 5.1 *Συναρτήσεις αλφαριθμητικών*

Στο παρόν κεφάλαιο θα μελετηθούν οι τέσσερις πρώτες συναρτήσεις του **Πίνακα 5.1**. Οι υπόλοιπες δύο συναρτήσεις θα μελετηθούν σε επόμενο κεφάλαιο, καθώς η ανάλυσή τους στηρίζεται στη χρήση δεικτών.

Παρατήρηση: Στις συνάρτησεις `stcpy()` και `strcat()` ελλοχεύει ο κίνδυνος να ξεπεραστούν τα όρια του πίνακα χαρακτήρων προορισμού, με αποτέλεσμα να δημιουργηθούν απροσδιόριστες καταστάσεις. Εάν π.χ. έχει οριστεί ο πίνακας χαρακτήρων `char array[4]`, ο οποίος καταλαμβάνει τις θέσεις **1000** έως και **1003** στον χάρτη μνήμης του **Σχήματος 5.2**, και επιχειρηθεί να τοποθετηθεί σε αυτό το αλφαριθμητικό

`"character"`

θα τεθεί θέμα απροσδιοριστίας. Συγκεκριμένα, στις θέσεις **1004-1009**, τις οποίες δεν διαχειρίζεται ο **`array`**, θα τοποθετηθούν οι χαρακτήρες '**a**', '**c**', '**t**', '**e**', '**r**', '**\0**'. Οι θέσεις, όμως, αυτές μπορεί να χρησιμοποιούνται από άλλες μεταβλητές και με την τροποποίηση του περιεχομένου τους οι νέες τιμές να προκαλέσουν σημασιολογικά σφάλματα στο πρόγραμμα.

Η εγγραφή δεδομένων εκτός των ορίων ενός πίνακα στην περιοχή προσωρινής αποθήκευσης ονομάζεται **υπερχείλιση της περιοχής προσωρινής αποθήκευσης** (buffer overflow). Για αυτόν τον λόγο θα πρέπει ο προγραμματιστής είτε να έχει προβλέψει επαρκή χώρο είτε σε κάθε χρήση αυτών των συναρτήσεων να εκτελεί έλεγχο ορίων, πριν τις χρησιμοποιήσει, όπως χαρακτηριστικά παρουσιάζεται στα παραδείγματα 5.9.2.1 και 5.9.3.1.

	διεύθυνση	τιμή
array[0]	1000	c
array[1]	1001	h
array[2]	1002	a
array[3]	1003	r
array[4]	1004	a
array[5]	1005	c
array[6]	1006	t
array[7]	1007	e
array[8]	1008	r
array[9]	1009	\0

Σχήμα 5.2 *Υπερχείλιση της περιοχής προσωρινής αποθήκευσης*

5.9.1. Η συνάρτηση εύρεσης μήκους αλφαριθμητικού

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

```
char stringName[20] = "My string";
printf( "%d\n", strlen(stringName) );
```

θα τυπώσει τον αριθμό των χαρακτήρων του αλφαριθμητικού **`stringName`**, δηλαδή **9** κι όχι **20**, που είναι ο αριθμός των στοιχείων του πίνακα χαρακτήρων **`stringName`**.

Μία υλοποίηση της **`strlen()`** ως συνάρτηση, η οποία θα δέχεται ως είσοδο το αλφαριθμητικό και θα επιστρέφει ως ακέραιο τον αριθμό των χαρακτήρων του αλφαριθμητικού, δίνεται ακολούθως:

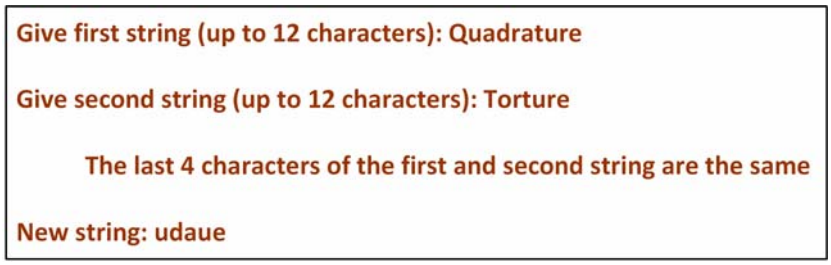
```
int strlenImplementation(char stringName[])
{
    int i=0;
    while (stringName[i]!='\0')  i++;
```

```
        return(i);
    }
```

5.9.1.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:
(α) Θα δέχεται από το πληκτρολόγιο δύο αλφαριθμητικά (μέγιστου μήκους 12) και θα τα αποθηκεύει.
(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>

    int main()
    {
         int i,j,sum;
         char str1[13],str2[13],str3[7];
         /* α */
         printf( "\nGive first string (up to 12 characters):  ");
         scanf("%s",str1);
         printf( "\nGive second string (up to 12 characters):  ");
         scanf("%s",str2);

         /* β */
         if ((strlen(str1)<4) || (strlen(str2)<4))
         printf( "\nError!! Strings should have more at least 4 charac-
ters!! Program aborted" );
         else
         {
            sum=0;
            for (i=0;i<4;i++)
               if (str1[strlen(str1)-i]==str2[strlen(str2)-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 are NOT the same\n " );
            }
         }
         /* γ */
         for (i=1;i<strlen(str1);i=i+2)  str3[i/2]=str1[i];
         str3[i/2]='\0'; /* για να καταστεί αλφαριθμητικό το str3 */
         printf( "\nNew string: %s",str3 );

         return 0;
      }
```



Εικόνα 5.12 *Η έξοδος του προγράμματος του παραδείγματος 5.9.1.1*

5.9.2. Η συνάρτηση αντιγραφής αλφαριθμητικού

Η συνάρτηση **`strcpy()`** (string copy) αντιγράφει ένα αλφαριθμητικό σε ένα άλλο. Δέχεται δύο ορίσματα που είναι τα ονόματα των αλφαριθμητικών. ***Ως πρώτο όρισμα τίθεται το αλφαριθμητικό προορισμού, ενώ ως δεύτερο όρισμα ορίζεται το προς αντιγραφή αλφαριθμητικό***. Στο παρακάτω τμήμα κώδικα

```
char str1[12] = "initial string";
char str[12] = "final string";
strcpy(str1,str2);
printf( "%s\n", str1 );
```

η πρόταση **`strcpy(str1,str2);`** αντιγράφει το περιεχόμενο του **`str2`** στον πίνακα χαρακτήρων **`str1`**. Έτσι, στην οθόνη θα εμφανιστεί η φράση **final string**.

Για να αντιγραφούν οι πρώτοι **`n`** χαρακτήρες του **`str2`** χρησιμοποιείται η σύνταξη **`strncpy(str1,str2,n)`**, όπου το τρίτο όρισμα είναι ο αριθμός των προς αντιγραφή χαρακτήρων.

Παρατήρηση: Η χρήση της συνάρτησης **`strcpy()`** αποτελεί τον τρόπο ανάθεσης ή εκχώρησης τιμής σε αλφαριθμητικό, καθώς η απευθείας ανάθεση τιμής επιστρέπεται μόνο στην αρχικοποίηση. Δηλαδή, η πρόταση

```
str1="Get a string";
```

δεν επιτρέπεται στη γλώσσα C. Η ανάθεση γίνεται μέσω της πρότασης:

```
strcpy(str1,"Get a string");
```

5.9.2.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:
(α) Θα δέχεται από το πληκτρολόγιο δύο αλφαριθμητικά και θα τα αποθηκεύει στον πίνακα αλφαριθμητικών **`str[2][41]`**. Κατά την ανάγνωση των αλφαριθμητικών θα γίνεται έλεγχος μέσω επαναληπτικής πρότασης, ώστε το μήκος τους να μην υπερβαίνει το **20**. Τα δοθέντα αλφαριθμητικά θα εμφανίζονται στην οθόνη.
(β) Τα δύο ανωτέρω αλφαριθμητικά θα μετασχηματίζονται με αφαίρεση των χαρακτήρων που βρίσκονται σε άρτιες θέσεις (π.χ. το αλφαριθμητικό **`"my string"`** θα μετασχηματιστεί σε **`"m tig"`**). Τα μετασχηματισμένα αλφαριθμητικά θα αποθηκεύονται στις ίδιες γραμμές του πίνακα και θα εμφανίζονται στην οθόνη.

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

int main()
{
  int i,j,length[2];
  char str[2][41],temp[21];
  /* (α) */
```

```
    for (i=0;i<2;i++)
    {
      do
      {
        printf( "\nGive string no %d (up to 20 characters):  ",i+1);
        gets(str[i]);
        length[i]=strlen(str[i]);
      }  while (length[i]>20);
    }
    printf("\n\n");
    for (i=0;i<2;i++) printf( "\nString no %d: '%s'",i+1,str[i] );

    /* (β) */
    for (i=0;i<2;i++)
    {
      for (j=0;j<length[i];j=j+2) temp[j/2]=str[i][j];
      temp[j/2]='\0';
      strcpy(str[i],temp);
      printf( "\nTransformed string no %d: '%s'",i+1,str[i] );
    }

    return 0;
}
```

```
Give string no 1 (up to 20 characters): my string

Give first string no 2 (up to 20 characters): your string


String no 1: 'my string'
String no 1: 'your string'
Transformed string no 1: 'm tig'
Transformed string no 1: 'yu tig'
```

Εικόνα 5.13 *Η έξοδος του προγράμματος του παραδείγματος 5.9.2.1*

5.9.3. Η συνάρτηση συνένωσης αλφαριθμητικών

Η συνάρτηση **`stcat()`** (string concatenation) 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 str1[30] = "first string";
char str2[30] = "second string";
strcat(str1,str2);
printf( "%sn", str1 );
```

προστίθεται στο τέλος του πίνακα χαρακτήρων **`str1`** το περιεχόμενο του πίνακα **`str2`**. Έτσι, στην οθόνη θα εμφανιστεί το αλφαριθμητικό **first stringsecond string**.

Για να προστεθούν οι πρώτοι **`n`** χαρακτήρες του **`str2`**, χρησιμοποιείται η σύνταξη **`strncat(str1,str2,n)`**, όπου το τρίτο όρισμα είναι ο αριθμός των προστιθέμενων χαρακτήρων.

5.9.3.1. Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:
(α) Θα δέχεται από το πληκτρολόγιο δύο αλφαριθμητικά και θα τα αποθηκεύει στον πίνακα αλφαριθμητικών `str[2][41]`. Τα δοθέντα αλφαριθμητικά θα εμφανίζονται στην οθόνη.
(β) Το δεύτερο αλφαριθμητικό θα συνενώνεται με το πρώτο, αφού προηγηθεί έλεγχος κατά πόσον υπάρχει διαθέσιμος χώρος για τη συνένωση. Εφόσον γίνει η συνένωση, το τροποποιημένο πρώτο αλφαριθμητικό θα εμφανίζεται στην οθόνη.

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

int main()
{
  char str[2][41];
  int i;

  for (i=0;i<2;i++)
  {
    printf( "\nGive string no %d (up to 40 characters):  ",i+1 );
    gets(str[i]);
  }
  printf("\n\n");

  if (strlen(str[0])+strlen(str[1])<=40)  strcat(str[0],str[1]);
  else
    printf( "\nERROR! The string cannot be concatenated." );

  printf( "\nConcatenated string: %s",str[0] );

  return 0;
}
```

```
Give string no 1 (up to 40 characters): first string

Give first string no 2 (up to 40 characters): second string


Concatenated string: first stringsecond string
```

Εικόνα 5.14 *Η έξοδος του προγράμματος του παραδείγματος 5.9.3.1*

5.9.4. Η συνάρτηση σύγκρισης αλφαριθμητικών

Η συνάρτηση `strcmp(str1,str2)` 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(str1,str2,n)`, όπου το τρίτο όρισμα είναι ο αριθμός των προς σύγκριση χαρακτήρων.

5.9.4.1. Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

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

#define N 4

int main()
{
  char msg1[41] = {"Quad"};
  char msg2[41]={"Quadrature"};
  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 );

  return 0;
}
```

```
First string=Quad
Second string=Quadrature
Different strings!
Same first 4 characters!
```

Εικόνα 5.15 *Η έξοδος του προγράμματος του παραδείγματος 5.9.4.1*

Δηλώνονται δύο πίνακες χαρακτήρων **41** θέσεων, στους οποίους αποδίδονται τα περιεχόμενα `"Quad"` και `"Quadrature"`, αντίστοιχα.

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

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποιες από τις ακόλουθες παρατηρήσεις, που αφορούν στους πολυδιάστατους πίνακες, είναι λανθασμένη;

(α) 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.

(β) Η 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 τον θεωρήσει είτε πίνακα 2x3 είτε 3x2.

(γ) Η πρόταση `printf( "%d",array[1,2] );` είναι λανθασμένη στη γλώσσα C αλλά δεν εντοπίζεται από τον μεταγλωττιστή και οδηγεί σε ανεπιθύμητα αποτελέσματα.
(δ) Η πρόταση `printf( "%d",array[1][2] );` είναι λανθασμένη στη γλώσσα C και εντοπίζεται από τον μεταγλωττιστή.

(2) Τι θα εμφανίσει στην οθόνη το ακόλουθο τμήμα κώδικα;

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

(α) **ef**
(β) **abcdef**
(γ) **efabcd**
(δ) **efcd**

(3) Ποιο πρόγραμμα ικανοποιεί τις ακόλουθες προδιαγραφές:
(i) Θα εισάγονται από το πληκτρολόγιο 4 αλφαριθμητικά σε πίνακα αλφαριθμητικών, μήκους 7 χαρακτήρων το καθένα.
(ii) Θα λαμβάνονται οι τρεις πρώτοι χαρακτήρες κάθε αλφαριθμητικού και θα συνενώνονται σε ένα νέο αλφαριθμητικό, το οποίο και θα τυπώνεται.

(α)
```
#include <stdio.h>
#include <string.h>
int main()
{
   int i;
   char str[4][7], str_total[13];
   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 );

   return 0;
}
```

(β)
```
#include <stdio.h>
int main()
{
   int i;
   char str[4][8], str_total[13];
   for (i=0;i<4;i++)
   {
      printf( "\nGive string no %d:  ",i+1 );
      scanf( "%s",str[i] );
      if (i==0) strncat(str_total,str[i],3);
      else strncat(str_total,str[i],3);
   }
   printf( "Total string: %s\n",str_total );

   return 0;
}
```

(γ)
```
#include <stdio.h>
#include <string.h>
int main()
{
   int i;
   char str[4][8], str_total[13];
   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 );

   return 0;
}
```
(δ)
```
#include <stdio.h>
#include <string.h>
int main()
{
   int i;
   char str[4][7], str_total[12];
   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 strncpy(str_total,str[i],3);
   }
   printf( "Total string: %s\n",str_total );

   return 0;
}
```

(4) Ποιο είναι το αποτέλεσμα της πρότασης `for (i=0,j=10; i<8; i++,j++) t[j]=s[i];`
(α) Αντιγραφή των οκτώ πρώτων στοιχείων του πίνακα **s** στον **t**, ξεκινώντας από το ενδέκατο στοιχείο του **t**.
(β) Αντιγραφή των οκτώ πρώτων στοιχείων του πίνακα **s** στον **t**.
(γ) Αντιγραφή των έντεκα πρώτων στοιχείων του πίνακα **s** στον **t** , ξεκινώντας από το όγδοο στοιχείο του **t**.
(δ) Αντιγραφή των τεσσάρων πρώτων στοιχείων του πίνακα **s** στον **t**, ξεκινώντας από το ενδέκατο στοιχείο του **t**.

(5) Ποιο είναι το αποτέλεσμα του ακόλουθου προγράμματος;
```
#include <stdio.h>
#define SIZE 10

int func(int b[], int p);

int main()
{
     int x;
     int a[SIZE]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
     x=func(a,SIZE);
     printf( "The result is %d\n", x );
```

```
        return 0;
    }

    int func(int b[], int p)
    {
        if (p==1) return(b[0]);
        else return(b[p-1]+func(b,p-1));
    }
```

(α) The result is 3628800
(β) The result is 55
(γ) The result is 433
(δ) The result is 0

Ασκήσεις

Άσκηση 1

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`).**

Άσκηση 2

Να γραφεί πρόγραμμα, το οποίο θα χρησιμοποιεί έναν πίνακα 1000 στοιχείων για να καθορίσει και να προβάλει τους πρώτους αριθμούς ανάμεσα στο 1 και το 999. Το στοιχείο 0 του πίνακα θα αγνοηθεί. Η υλοποίηση στηρίζεται στα παρακάτω:

- **Πρώτος αριθμός καλείται οποισδήποτε ακέραιος μπορεί να διαιρεθεί μόνο με τον εαυτό του και το 1.**
- **Ο αλγόριθμος υπολογισμού ονομάζεται *κόσκινο (ή κρησάρα) του Ερατοσθένους* και λειτουργεί ως εξής:**
  - **Δημιουργείται ένας πίνακας με όλα τα στοιχεία να έχουν τιμή 1. Τα στοιχεία του πίνακα με αριθμοδείκτη πρώτο αριθμό θα διατηρήσουν την τιμή 1. Τα υπόλοιπα θα αποκτήσουν σταδιακά την τιμή 0.**
  - **Ξεκινώντας από τον αριθμοδείκτη 2 (θεωρώντας πρώτο αριθμοδείκτη το 1 και όχι το 0), κάθε φορά που βρίσκεται ένα στοιχείο του πίνακα με τιμή 1 (έστω ότι αντιστοιχεί στον αριθμοδείκτη `k`), ενεργοποιείται ένας βρόχος, κατά τη διάρκεια του οποίου μηδενίζονται όλα τα στοιχεία του πίνακα που έχουν αριθμοδείκτες πολλαπλάσιους του `k`. Π.χ. για τον αριθμοδείκτη 2 θα μηδενιστούν οι θέσεις του πίνακα με αριθμοδείκτες 4,6,8,10 κ.λ.π., ενώ για τον αριθμοδείκτη 3 θα μηδενιστούν οι θέσεις 6,9,12,15 κ.λπ. Όταν ολοκληρωθεί η διαδικασία, μη μηδενικές τιμές θα έχουν μόνο τα στοιχεία που βρίσκονται σε θέσεις με αριθμοδείκτες πρώτους αριθμούς.**

Άσκηση 3

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- **Μέσω επαναληπτικής πρότασης θα λαμβάνει από το πληκτρολόγιο τις τιμές ενός πίνακα τεσσάρων αλφαριθμητικών `arr_char[4][26]`. Τα αλφαριθμητικά θα δύνανται να περιέχουν τον χαρακτήρα του κενού. Ακολούθως, θα εμφανίζει τον προκύπτοντα πίνακα στην οθόνη.**
- **Θα σαρώνει τον πίνακα ελέγχοντας εάν σε κάθε αλφαριθμητικό εμφανίζεται τουλάχιστον 3 φορές ο χαρακτήρας `'D'`. Σε κάθε επιτυχή έλεγχο, θα αυξάνεται κατάλληλος μετρητής. Στο τέλος της σάρωσης του πίνακα θα εμφανίζονται στην οθόνη η τιμή του μετρητή επιτυχών ελέγχων και οι θέσεις στον πίνακα `arr_char` όπου εμφανίστηκε τουλάχιστον 3 φορές ο χαρακτήρας `'D'`.**

Άσκηση 4

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- **Μέσω επαναληπτικής πρότασης `do-while` θα δέχεται από το πληκτρολόγιο τρία αλφαριθμητικά και θα τα αποθηκεύει σε πίνακα αλφαριθμητικών `str[3][21]`. Ακολούθως, θα εμφανίζει στην οθόνη τα περιεχόμενα του πίνακα.**
- **Θα ελέγχει εάν οι χαρακτήρες που βρίσκονται στην πρώτη, δεύτερη και τρίτη θέση του του πρώτου αλφαριθμητικού είναι ίδιοι με τους χαρακτήρες που βρίσκονται στις αντίστοιχες θέσεις του δεύτερου αλφαριθμητικού (ελέγχοντας πρώτα εάν τα αναγνωσθέντα αλφαριθμητικά έχουν μήκος μεγαλύτερο ή ίσο του 3). Σε περίπτωση ισότητας θα αντιγράφει στην τρίτη γραμμή του πίνακα τους 3 αυτούς χαρακτήρες και θα τυπώνει το νέο πίνακα χαρακτήρων στην οθόνη ως αλφαριθμητικό.**

Άσκηση 5

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- **Θα λαμβάνει από το πληκτρολόγιο τις τιμές ενός πίνακα δύο αλφαριθμητικών `arr_char[2][15]` και θα εμφανίζει τον προκύπτοντα πίνακα στην οθόνη.**
- **Θα δημιουργεί στην πρώτη γραμμή του πίνακα `arr_char` (δηλαδή στην `arr_char[0]`) το αλφαριθμητικό που θα προκύψει αποκλειστικά από τους χαρακτήρες του `arr_char[1]`, οι οποίοι βρίσκονται στις περιττές θέσεις (υπενθυμίζεται ότι η πρώτη θέση πίνακα στη γλώσσα C είναι άρτια, ήτοι η θέση 0).**

Άσκηση 6

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- **Θα λαμβάνει από το πληκτρολόγιο τις τιμές ενός πίνακα ακέραιων αριθμών `arr_int[3][3]` και θα εμφανίζει τον προκύπτοντα πίνακα στην οθόνη.**
- **Θα υπολογίζει σε κάθε γραμμή του πίνακα `arr_int` το στοιχείο με την ελάχιστη απόλυτη τιμή. Για κάθε γραμμή θα απεικονίζονται στην οθόνη η απόλυτη τιμή του ελάχιστου στοιχείου και η στήλη στην οποία βρίσκεται.**
- **Θα υπολογίζει το γινόμενο των τιμών των στοιχείων της δεύτερης στήλης του πίνακα `arr_int`.**

Άσκηση 7

Να γραφεί πρόγραμμα, με το οποίο θα εισάγονται 9 ακέραιοι αριθμοί σε τετραγωνικό πίνακα `array[3][3]`. Ακολούθως, θα καλείται η συνάρτηση `float aver(int arr[3][3], int size)`, η οποία θα επιστρέφει στη `main()` τον μέσο όρο των τιμών των στοιχείων του πίνακα, ο οποίος και θα εμφανίζεται στην οθόνη.

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Deitel, H. & Deitel, P. (2005), *Ασκήσεις - Προγράμματα σε C*, Εκδόσεις Γκιούρδα.

Kelley, A. & Pohl, I. (1998), *A Book on C*, 4th ed , Addison-Wesley.

King, K. (2008), *C Programming: A Modern Approach*, 2nd ed., W.W. Norton & Company.

Prinz, P. & Crawford, T. (2005), *C in a Nutshell*, O'Reilly.

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

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

6. Δείκτες

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια του δείκτη. Αρχικά, δίνεται ο ορισμός και τα χαρακτηριστικά του δείκτη, η δήλωσή του και οι τρόποι ανάθεσης τιμής. Ακολούθως, μελετάται ο χειρισμός μεταβλητών και πινάκων με χρήση δεικτών. Στην επόμενη ενότητα παρουσιάζονται τα ζητήματα της κλήσης συνάρτησης κατ' αναφορά και των συναρτήσεων με επιστρεφόμενο τύπο δείκτη. Στη συνέχεια, περιγράφονται οι υλοποιήσεις συναρτήσεων αλφαριθμητικών με χρήση δεικτών.Το κεφάλαιο ολοκληρώνεται με τις έννοιες των ορισμάτων της γραμμής εντολών και του δείκτη σε συνάρτηση.

Λέξεις κλειδιά
δείκτης, κλήση κατ' αναφορά, διεύθυνση μεταβλητής και πίνακα, τελεστής διεύθυνσης, τελεστής περιεχομένου, συνάρτηση strchr, συνάρτηση strstr, ορίσματα γραμμής εντολών, δείκτης σε συνάρτηση.

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

6.1 Η έννοια του δείκτη

Σε κάθε μεταβλητή αποδίδεται 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 αποθηκευτούν ή να ανακληθούν δεδομένα. Αντίθετα, στις γλώσσες προγραμματισμού υψηλού επιπέδου οι διευθύνσεις δεν είναι άμεσα ορατές από τον προγραμματιστή, καθώς καλύπτονται από τον 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 ιδιάζουσα μορφή μεταβλητής, η οποία έχει ως περιεχόμενο όχι ένα πραγματικό δεδομένο αλλά μία διεύθυνση μνήμης***. Οι δείκτες είναι ένα ισχυρό προγραμματιστικό εργαλείο και εφαρμόζονται:

- 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.

6.2 Δήλωση δείκτη

Η δήλωση ενός δείκτη ακολουθεί τον εξής φορμαλισμό:

```
<τύπος δεδομένων> *<όνομα δείκτη>;
```

π.χ.

```
int *pnum;
```

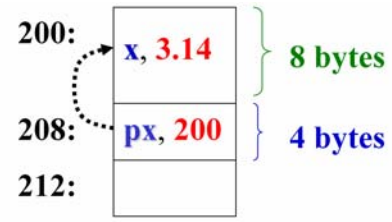
Όταν ένας δείκτης έχει αποθηκευμένη μία διεύθυνση, έχει επικρατήσει να λέγεται ότι *ο δείκτης «δείχνει» στη διεύθυνση*. Στη δήλωση δείκτη ο **<τύπος δεδομένων>** αφορά στο είδος των δεδομένων που αποθηκεύονται στη διεύθυνση που «δείχνει» ο δείκτης. Ο τύπος της κανονικής μεταβλητής πρέπει να δηλώνεται, γιατί μία μεταβλητή δεσμεύει συγκεκριμένη μνήμη (8 bytes για **double**, 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 ένας δείκτης κι όχι μία κανονική μεταβλητή. Ο αστερίσκος συνδέεται με το όνομα και όχι με τον τύπο δεδομένων. Θα πρέπει να σημειωθεί ότι αν και ο δείκτης περιέχει μία διεύθυνση (δηλαδή έναν ακέραιο αριθμό), δεν είναι ίδιος με μία κανονική ακέραια μεταβλητή και δεν ακολουθεί την αριθμητική ακεραίων. Μέσω του αστερίσκου ο μεταγλωττιστής γνωρίζει ότι η τιμή του 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 δεικτών σε ακέραιες μεταβλητές και σε μεταβλητές τύπου χαρακτήρα:

```
int *pcount,*count_ptr;
char *pword,*word_ptr;
```

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 6.1** απεικονίζεται ο τρόπος λειτουργίας των δεικτών, με τη **x** να είναι μία κανονική μεταβλητή τύπου **double** και τον **px** να είναι δείκτης που δείχνει στη **x**.



Σχήμα 6.1 *Απεικόνιση του τρόπου λειτουργίας των δεικτών*

6.3 Ανάθεση τιμής σε δείκτη

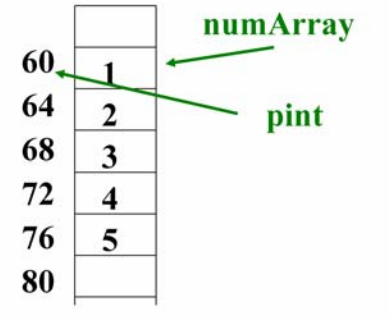
Η ανάθεση τιμής σε δείκτη μπορεί να γίνει με έναν από τους ακόλουθους τέσσερις τρόπους:
(1) ***Με χρήση του τελεστή διεύθυνσης*** (**&**) (address-of operator). Ο τελεστής διεύθυνσης **&**, ο οποίος πρωτοαπαντήθηκε στη συνάρτηση εισόδου ***scanf()***, εισάγεται μπροστά από μία μεταβλητή εκφράζοντας τη διεύθυνσή της. Ο συμβολισμός **&count** ερμηνεύεται «στη διεύθυνση της **count**». Έτσι, με τον ακόλουθο κώδικα ανατίθεται στον δείκτη **pnum** η διεύθυνση, στην οποία βρίσκεται αποθηκευμένη η ακέραια μεταβλητή **count**.

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

(2) ***Με χρήση πινάκων***, δεδομένου ότι το όνομα ενός πίνακα αντιστοιχεί στη διεύθυνση του πρώτου στοιχείου του. Έτσι, με τον ακόλουθο κώδικα αποδίδεται στον δείκτη ακεραίων **pint** η τιμή **60**, δηλαδή η

διεύθυνση του πρώτου στοιχείου του πίνακα **numArray**. Στο **Σχήμα 6.2** απεικονίζεται η διαδικασία ανάθεσης τιμής στον δείκτη **pint**.

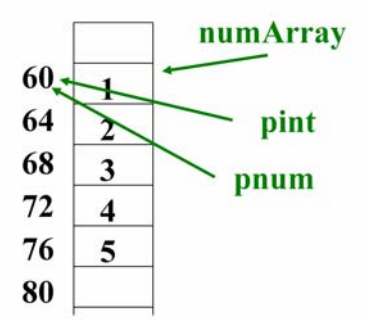
```
int numArray[5]={1,2,3,4,5};
int *pint;
pint=numArray;
```



Σχήμα 6.2 *Ανάθεση τιμής σε δείκτη με χρήση πίνακα*

(3) ***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 6.3**.

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



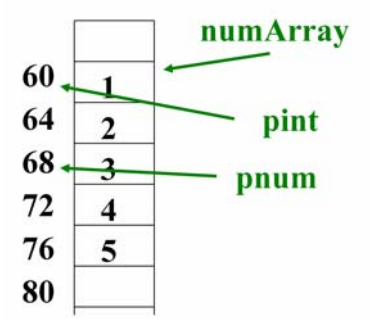
Σχήμα 6.3 *Ανάθεση τιμής σε δείκτη με χρήση δεικτών ίδιου τύπου*

(4) ***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 έγκειται στο ότι στον **pnum** δεν 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 6.4**.

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



Σχήμα 6.4 *Ανάθεση τιμής σε δείκτη με χρήση αριθμητικής δεικτών*

Παρατήρηση: Θεωρητικά, μπορεί να γίνει άμεση ανάθεση μίας διεύθυνσης, π.χ. `pnum=1001;`. Ωστόσο, μία τέτοια επιλογή είναι πολύ επικίνδυνη και πρέπει πάντοτε να αποφεύγεται, καθώς κατά την εκτέλεση της γραμμής κώδικα δεν μπορεί να είναι γνωστό κατά πόσον οι διευθύνσεις από **1001** έως **1004** (δηλαδή τα 4 bytes που καταλαμβάνει ο ακέραιος, στον οποίο θα δείχνει ο `pnum`) είναι κατειλημμένες. Τέτοιου είδους αναθέσεις χρησιμοποιούνται συνήθως μόνο για άμεση πρόσβαση στο υλικό (hardware).

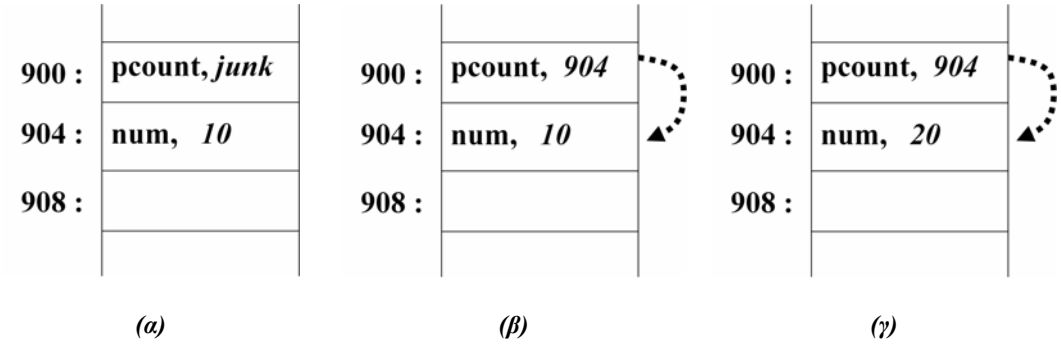
6.4 Προσπέλαση μεταβλητής και πίνακα με χρήση δείκτη

Η δαιχείριση μίας μεταβλητής μέσω δείκτη γίνεται με χρήση του **τελεστή περιεχομένου** (`*`) (dereferencing operator) ή **τελεστή έμμεσης αναφοράς** ή **έμμεσης διευθυνσιοδότησης**, η λειτουργία του οποίου περιγράφεται με τη βοήθεια του ακόλουθου κώδικα:

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

Στον παραπάνω κώδικα ορίζεται μία ακέραια μεταβλητή `num`, η οποία λαμβάνει την τιμή **10**, και ένας δείκτης σε ακέραιο `pcount`, ο οποίος αρχικά δεν έχει τιμή. Στο **Σχήμα 6.5.α** παρουσιάζεται ο χάρτης μνήμης μετά το τέλος της δεύτερης γραμμής κώδικα. Ως ***junk*** (απορρίματα) συμβολίζεται το περιεχόμενο μίας θέσης μνήμης, όταν αυτό δεν έχει καθοριστεί από το πρόγραμμα.

Ακολούθως, ανατίθεται στον δείκτη `pcount` η διεύθυνση της `num` (**Σχήμα 6.5.β**). Στην τελευταία γραμμή κώδικα χρησιμοποιείται ο τελεστής περιεχομένου μπροστά από τον 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 6.5.γ**).



Σχήμα 6.5 *Προσπέλαση μεταβλητής με χρήση δείκτη*

Οι δείκτες χρησιμοποιούνται και για την προσπέλαση πινάκων. Μάλιστα, στο επόμενο κεφάλαιο, όπου θα μελετηθεί η δυναμική διαχείριση μνήμης, η δομή του πίνακα θα υλοποιηθεί αποκλειστικά με χρήση δείκτη. Έστω ένας πίνακας ακεραίων `arr[5]`, ο οποίος αποθηκεύεται στη μνήμη στα bytes **1000** έως και **1019**, και ο δείκτης σε ακέραιο `parr`. Με την πρόταση `parr=arr;` ο δείκτης `parr` μπορεί πλέον να προσπελάσει τα δεδομένα του πίνακα, καθώς το όνομα ενός πίνακα αντιστοιχεί στο πρώτο byte της θέσης μνήμης που καταλαμβάνει το πρώτο στοιχείο του πίνακα. Με την ακόλουθη σημειογραφία:

`parr[i]=...` ή `*(parr+i)=...`

ο δείκτης αναθέτει με έμμεσο τρόπο τιμή (ή γενικά χειρίζεται) στο στοιχείο `arr[i]` του πίνακα `arr`. Κατά συνέπεια, μακροσκοπικά ο χειρισμός των στοιχείων του πίνακα με χρήση δείκτη φαίνεται απολύτως αντίστοιχος με τον χειρισμό μέσω μεταβλητής πίνακα. Η διαφορά έγκειται στο ότι ο δείκτης λειτουργεί έμμεσα, προσπελαύνοντας την *i*-στη θέση μνήμης (στο συγκεκριμένο παράδειγμα και με βάση την αριθμητική δεικτών) τα bytes **1000+(4*i)** έως **1000+(4*i)+3**, ενώ η μεταβλητή πίνακα προσπελαύνει άμεσα το στοιχείο `arr[i]`.

Η έκφραση `parr=arr;` είναι ισοδύναμη με την έκφραση `parr=&arr[0];`.

6.4.1 Παράδειγμα

Να περιγραφεί αναλυτικά η λειτουργία κάθε γραμμής κώδικα και να απεικονιστούν 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`).

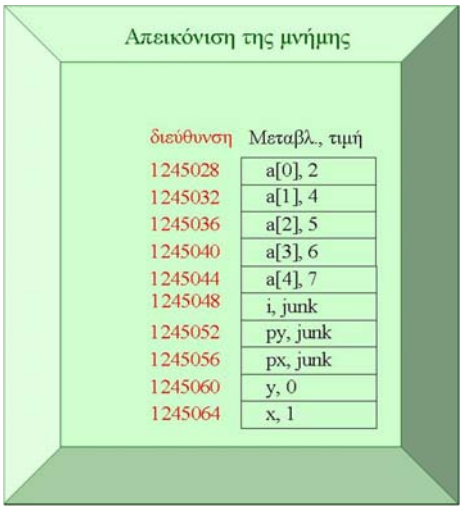
Γραμμή 2: Δήλωση πίνακα ακεραίων πέντε στοιχείων (`a`) και αρχικοποίηση αυτού.

Γραμμή 3: Δήλωση ακέραιας μεταβλητής (`i`) (**Σχήμα 6.6.α**).

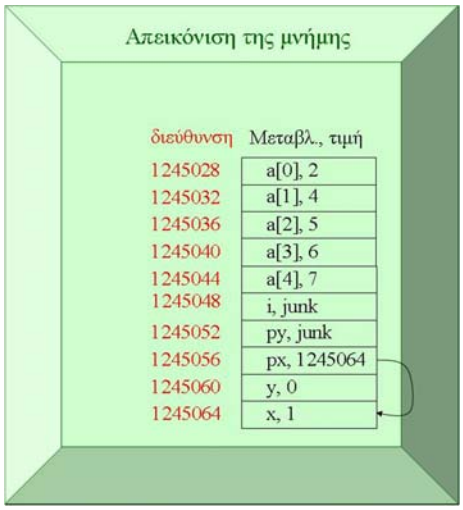
Γραμμή 4: Ανάθεση της διεύθυνσης της μεταβλητής `x` στον δείκτη `px`, δηλαδή το περιεχόμενο του `px` είναι η διεύθυνση – το πρώτο byte – του `x` (**Σχήμα 6.6.β**).

Γραμμή 5: Ανάθεση της διεύθυνσης του πρώτου στοιχείου του πίνακα `a` στον δείκτη `py`, δηλαδή το περιεχόμενο του `py` είναι η διεύθυνση του `a[0]` (**Σχήμα 6.6.γ**).

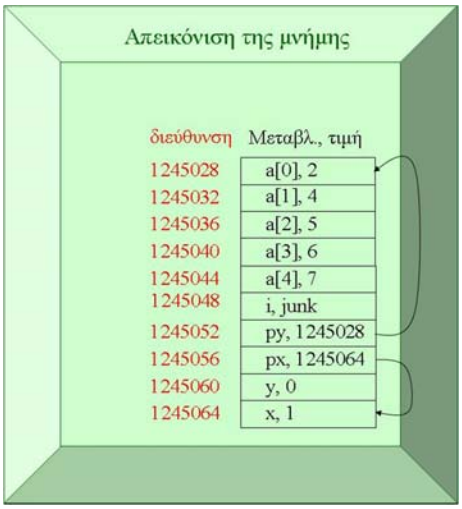
Γραμμή 6: Επαναληπτική έκφραση, σε κάθε επανάληψη της οποίας λαμβάνουν χώρα τα ακόλουθα: η τιμή **2*i** τοποθετείται σε θέση μνήμης, η οποία βρίσκεται σε διεύθυνση **2*i** bytes μετά τη διεύθυνση που δείχνει ο `py` (**Σχήματα 6.6.δ–6.6.η**). Για παράδειγμα, εάν `i=3`, η τιμή **6** αποθηκεύεται στη θέση μνήμης **py+i=1245028+3*4=1245028+12=1245040**, θέση που καταλαμβάνει το στοιχείο `a[3]`. Με αυτόν τον τρόπο αποδίδεται στο `a[3]` η τιμή **6**.



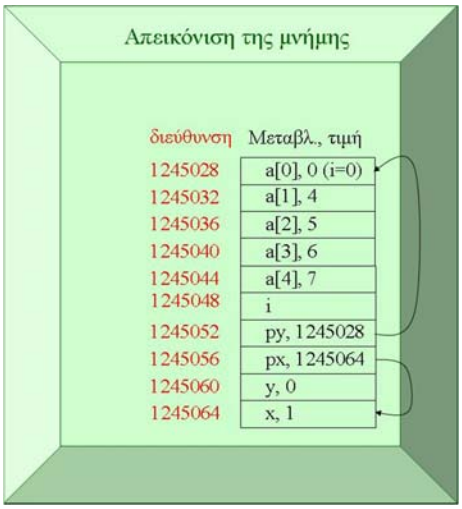
Σχήμα 6.6.α



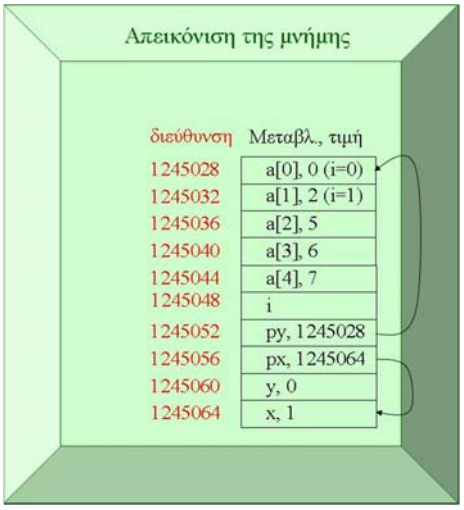
Σχήμα 6.6.β



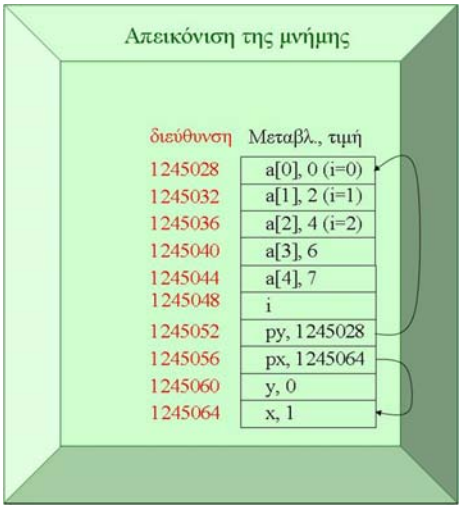
Σχήμα 6.6.γ



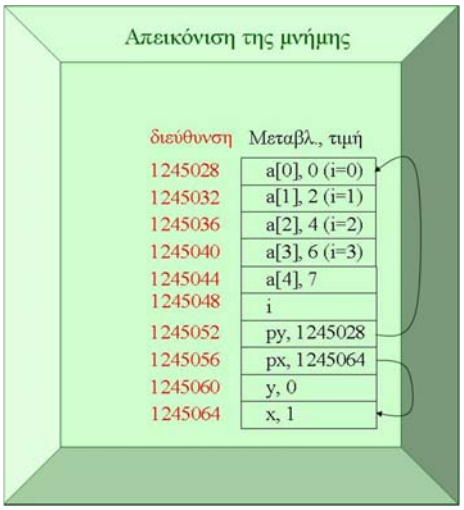
Σχήμα 6.6.δ



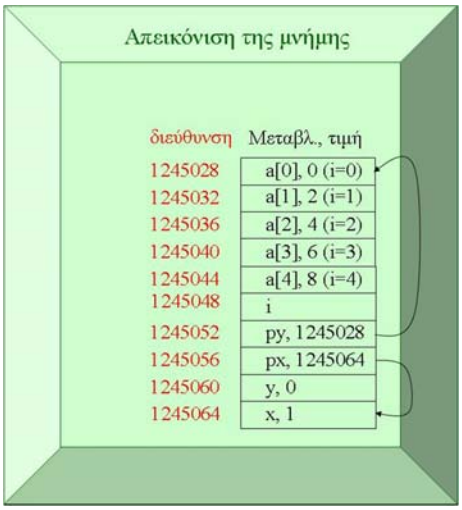
Σχήμα 6.6.ε



Σχήμα 6.6.στ



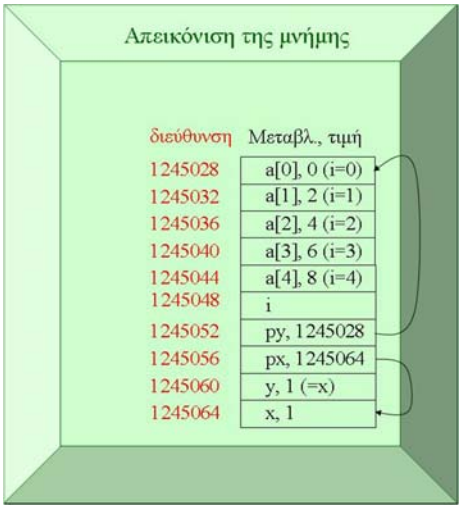
Σχήμα 6.6.ζ



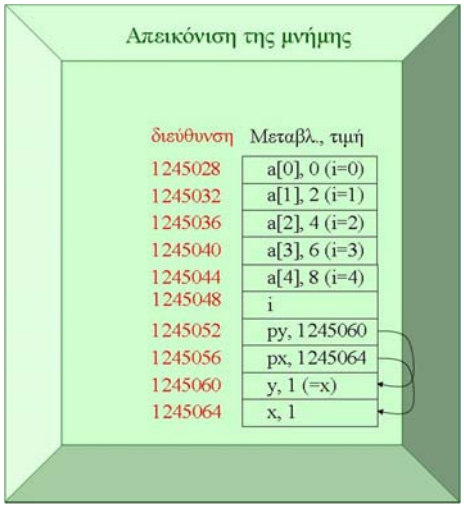
Σχήμα 6.6.η

Γραμμή 7: Το περιεχόμενο της θέσης στην οποία δείχνει ο `px`, δηλαδή το **1**, αποδίδεται στη μεταβλητή `y` (**Σχήμα 6.6.θ**).

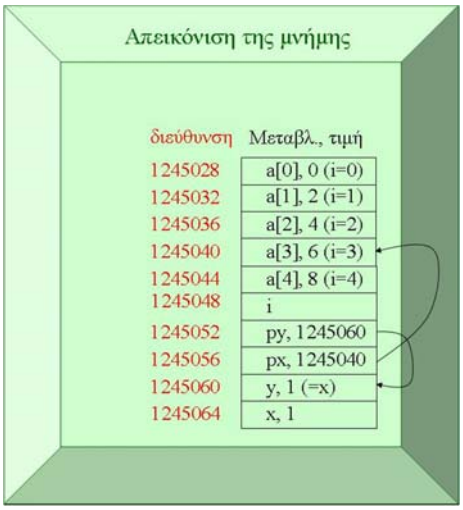
Γραμμή 8: Ανάθεση της διεύθυνσης της μεταβλητής `y` στον δείκτη `py` (**Σχήμα 6.6.ι**).



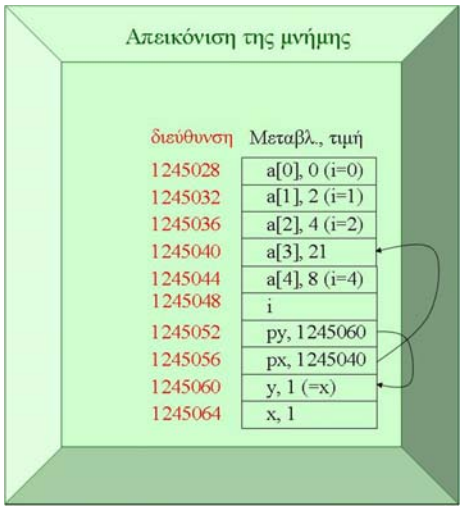
Σχήμα 6.6.θ



Σχήμα 6.6.ι



Σχήμα 6.6.ια



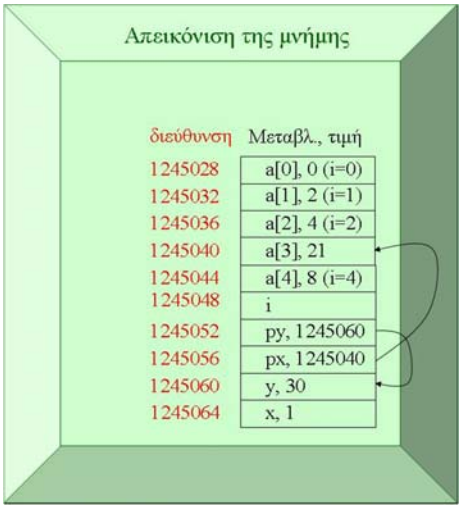
Σχήμα 6.6.ιβ

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 6.6.ια**).

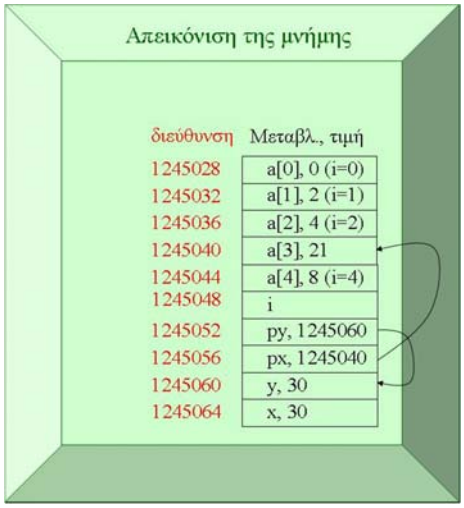
Γραμμή 10: Το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο **px** γίνεται **21** (**Σχήμα 6.6.ιβ**).

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 6.6.ιγ**).

Γραμμή 12: Η τιμή της μεταβλητής **x** γίνεται ίση με την τιμή της μεταβλητής **y** (**Σχήμα 6.6.ιδ**).



Σχήμα 6.6.ιγ



Σχήμα 6.6.ιδ

6.5 Δείκτες ως παράμετροι ή ως επιστρεφόμενοι τύποι συνάρτησης

6.5.1 Μεταβίβαση παραμέτρων – κλήση κατ' αναφορά

Στο Κεφάλαιο 4 αναλύθηκε 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, 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 *δεν 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 τις διευθύνσεις των μεταβλητών ως πραγματικά ορίσματα και δείκτες ως τυπικές παραμέτρους.

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 byte 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, οι οποίες είναι δείκτες, αντιγράφουν μία διεύθυνση στις παραμέτρους της συνάρτησης, αλλά εάν αλλαχθεί η παράμετρος στη συνάρτηση (δηλαδή η διεύθυνση), δεν θα αλλαχθεί η πραγματική παράμετρος! Το ακόλουθο παράδειγμα αναδεικνύει το πρόβλημα.

6.5.1.1 Παράδειγμα

Να περιγραφεί αναλυτικά ο ακόλουθος κώδικας:

```
#include <stdio.h>

void print(int *ptr);

int main()
{
  int *pscore, num;
  num=32;
  pscore=&num;
  printf( "Prior to function execution,  *pscore=%d\n",*pscore );
  print(pscore);
  printf( "After function execution,  *pscore=%d\n",*pscore );

  return 0;
}

void print(int *ptr)
{
  printf( "During function execution,  *ptr=%d\n", *ptr );
  ptr++;
}
```

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 ακολούθως:

```
Prior to function execution,  *pscore=32
During function execution,  *ptr=32
After function execution,  *pscore=32
```

Εικόνα 6.1 *Η έξοδος του προγράμματος του παραδείγματος 6.5.1.1*

6.5.1.2 Παράδειγμα

Να γίνει συγκριτική ανάλυση της λειτουργίας των ακόλουθων προγραμμάτων:

```
/* 1ο πρόγραμμα */
#include <stdio.h>

void swap(int a, int b);

int main()
{
   int x=10, y=25;
   printf( "Prior to function execution:  x=%d, y=%d\n", x, y );
```

```
   swap(x,y);
   printf( "After function execution:  x=%d, y=%d\n", x, y );

   return 0;
}

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

```
/* 2ο πρόγραμμα */
#include <stdio.h>

void swap(int *pa, int *pb);

int main()
{
   int x=10, y=25;
   printf( "Prior to function execution:  x=%d, y=%d\n", x, y );
   swap(&x, &y);
   printf( "After function execution:  x=%d, y=%d\n", x, y );

   return 0;
}

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

Τα παραπάνω προγράμματα καλούν τη συνάρτηση **swap()** με σκοπό την αντιμετάθεση των περιεχομένων των μεταβλητών **x** και **y**. Στο πρώτο 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 συνάρτησης **swap()** τα **a** και **b** να μην είναι πλέον ενεργά και οι τιμές των **x** και **y** να έχουν παραμείνει αμετάβλητες. Κατά συνέπεια, το πρώτο πρόγραμμα δεν εκτέλεσε επιτυχώς το έργο της αντιμετάθεσης, όπως άλλωστε προκύπτει και από τα αποτελέσματα:

```
Prior to function execution:  x=10, y=25
After function execution:  x=10, y=25
```

Εικόνα 6.2.α *Η έξοδος του πρώτου προγράμματος του παραδείγματος 6.5.1.2*

Στο δεύτερο πρόγραμμα χρησιμοποιείται η κλήση κατ' αναφορά και οι διευθύνσεις των μεταβλητών **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 μνήμης που αντιστοιχεί στη μεταβλητή **x**, αποκτά το περιεχόμενο της θέσης μνήμης, που αντιστοιχεί στη μεταβλητή **y** και αντίστροφα. Μετά το πέρας της συνάρτησης **swap()** παύουν να υφίστανται οι δείκτες **pa** και **pb** και το έργο της αντιμετάθεσης έχει επιτευχθεί, όπως φανερώνουν και τα αποτελέσματα:

Εικόνα 6.2.β *Η έξοδος του δεύτερου προγράμματος του παραδείγματος 6.5.1.2*

6.5.1.3 Παράδειγμα

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

```
#include <stdio.h>

void f1(char *pe);

int main()
{
  int a=8,*pb;
  f1("gnimmargorP larudecorP");
  pb=&a;  a=14;   *pb=13;
  printf( "\n\n\ta=%d\n",a );

  return 0;
}

void f1(char *pe)
{
  char *ps;
  ps=pe;
  while (*pe)  pe++;
  do
  {
    pe--;
    printf( "%c",*pe );
  } while(pe>ps);
}
```

- Στη συνάρτηση **main()** ορίζονται η ακέραια μεταβλητή **a**, στην οποία δίνεται αρχική τιμή **8**, και ο δείκτης σε ακέραιο **pb**. Στη συνέχεια, καλείται η συνάρτηση **f1()** με όρισμα τη συμβολοσειρά **"gnimmargorP larudecorP"**, η οποία αποδίδεται στον δείκτη σε χαρακτήρα **pe**, δηλαδή ο **pe** θα δείχνει στη διεύθυνση του πρώτου χαρακτήρα της συμβολοσειράς, **'g'**.
- Μέσα στη συνάρτηση **f1()** ορίζεται ως τοπική μεταβλητή ο δείκτης **ps**, στον οποίο αποδίδεται το περιεχόμενο του **pe**, επομένως ο **ps** θα δείχνει στον χαρακτήρα **'g'**. Ακολούθως, εκτελείται ένας βρόχος **while**, ο οποίος έχει ως συνθήκη το περιεχόμενο της θέσης μνήμης να είναι διάφορο του μηδενικού χαρακτήρα. Επομένως, ο βρόχος θα εκτελεστεί τόσες φορές όσες είναι το μήκος της συμβολοσειράς (**22** φορές). Σε κάθε επανάληψη του βρόχου ο δείκτης **pe** μετακινείται κατά ένα byte παρακάτω. Στο τέλος του βρόχου ο **pe** θα δείχνει στον μηδενικό χαρακτήρα τερματισμού της συμβολοσειράς **"gnimmargorP larudecorP"**.
- Ακολουθεί ένας βρόχος **do-while**, στον οποίο ο δείκτης **pe** μεταφέρεται ένα byte ψηλότερα και στη συνέχεια τυπώνεται το περιεχόμενο της διεύθυνσης στην οποία δείχνει. Ο βρόχος διαρκεί όσο ισχύει η συνθήκη **pe>ps**. Όταν η συνθήκη καταστεί ψευδής, θα έχει εκτυπωθεί η συμβολοσειρά αντεστραμμένη, δηλαδή **Procedural Programming**. Το τέλος του βρόχου οδηγεί και στον τερματισμό της κλήσης της συνάρτησης **f1()** και ο έλεγχος του προγράμματος επιστρέφει στη γραμμή

```
pb=&a;  a=14;  *pb=13;
```

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

6.5.1.4 Παράδειγμα

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.

(γ) 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);

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

  return 0;
```

```
}

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;
}
```

```
Give a number: 1407
Coded number=7481

Give a number: 0000
Coded number=7777

Give a number: 5555
Coded number=2222
```

Εικόνα 6.3 *Η έξοδος του προγράμματος του παραδείγματος 6.5.1.4*

6.5.2 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, τα οποία θα παραμένουν ενεργά και μετά το πέρας της καλούμενης συνάρτησης (π.χ. να δείχνει σε δεδομένα της καλούσας συνάρτησης). Δεν πρέπει ποτέ να επιστρέφει δείκτης που δείχνει σε τοπική μεταβλητή της καλούμενης συνάρτησης, γιατί όταν τερματιστεί η συνάρτηση, οι τοπικές μεταβλητές εξαφανίζονται (εξαιρούνται οι static τοπικές μεταβλητές, για τις οποίες θα γίνει μνεία παρακάτω).

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**.

6.5.2.1 Παράδειγμα

Θεωρούμε τον ακόλουθο κώδικα:

```
int *incr();

int main()
{
  int *pscore;
  pscore=incr();
  printf( "pscore point to %d",pscore );

  return 0;
}

int *incr()
{
  int y;
  y=10;
  return(&y);
}
```

Στον παραπάνω κώδικα υπάρχει σφάλμα, γιατί όταν τελειώνει η συνάρτηση **incr()**, η τοπική μεταβλητή **y** εξαφανίζεται και η τιμή της χάνεται. Επιστρέφοντας ο έλεγχος του προγράμματος στη **main()**,ο δείκτης **pscore** θα δείχνει στη διεύθυνση που επέστρεψε από τη συνάρτηση **incr()**, ωστόσο θα υπάρχει «σκουπίδι» (junk) σε αυτή τη διεύθυνση, εφόσον το **y** έχει παύσει να ισχύει. Μάλιστα, κατά τη μεταγλώττιση οι διάφοροι μεταγλωττιστές παρέχουν σχετικό μήνυμα προειδοποίησης (warning) για «ύποπτη μετατροπή δείκτη» (suspicious pointer conversion) ή «η συνάρτηση επιστρέφει διεύθυνση τοπικής μεταβλητής» (function returns address of local variable). 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()**.

6.6 Δείκτες και συναρτήσεις αλφαριθμητικών

Στο Κεφάλαιο 5 μελετήθηκαν ορισμένες συναρτήσεις διαχείρισης αλφαριθμητικών. Στην παρούσα ενότητα θα περιγραφούν δύο ακόμη συναρτήσεις και θα δοθούν υλοποιήσεις με χρήση δεικτών.

6.6.1 Η συνάρτηση εύρεσης χαρακτήρα σε αλφαριθμητικό

Η συνάρτηση **strchr(str1,ch)** αναζητά μέσα στο αλφαριθμητικό **str1** τον χαρακτήρα που περιέχει η μεταβλητή **ch**. Στην πρώτη εύρεση του χαρακτήρα η συνάρτηση επιστρέφει ένα δείκτη σε χαρακτήρα. Εάν δεν βρεθεί ο χαρακτήρας, επιστρέφεται η μηδενική διεύθυνση.

Η συνάρτηση **strchr()** δέχεται δύο ορίσματα: το πρώτο όρισμα είναι το όνομα του αλφαριθμητικού και το δεύτερο όρισμα είναι ο χαρακτήρας. Η συνάρτηση **strchr()** ορίζεται στο αρχείο κεφαλίδας **string.h**.

Θα πρέπει να σημειωθεί ότι ο δείκτης που επιστρέφει η συνάρτηση, ουσιαστικά αντιστοιχεί στο τμήμα του αλφαριθμητικού που αρχίζει από τον υπό εύρεση χαρακτήρα, όπως φαίνεται στο παράδειγμα που ακολουθεί.

6.6.1.1 Παράδειγμα

Στο παρακάτω πρόγραμμα αναζητάται ο χαρακτήρας **'a'** μέσα στο αλφαριθμητικό **"triangular"**.

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

int main()
{
  char msg1[81]={"triangular"};
  char *pfnd;
  printf( "\nmsg1: %s\n",msg1 );
  printf( "Address of string msg1: %d\n",msg1 );
  pfnd = strchr(msg1,'a');
  printf( "\npfnd: %s\n",pfnd );
  printf( "Address of string handled by pointer pfnd: %d\n",pfnd );

  return 0;
}
```

```
msg1: triangular
Address of string msg1: 2358752

pfnd: angular
Address of string handled by pointer pfnd: 2358755
```

Εικόνα 6.4 *Η έξοδος του προγράμματος του παραδείγματος 6.6.1.1*

Από τα αποτελέσματα της εκτέλεσης του προγράμματος συνάγεται ότι ο δείκτης **pfnd** λαμβάνει ως τιμή το byte, στο οποίο εμφανίζεται για πρώτη φορά ο χαρακτήρας **'a'** (byte **2358755**). Αυτό σημαίνει ότι ο δείκτης **pfnd** μπορεί να χειριστεί ένα νέο αλφαριθμητικό, το οποίο είναι το τμήμα του αλφαριθμητικού **msg1** από το ανωτέρω byte έως το τέλος του, δηλαδή το αλφαριθμητικό **"angular"**.

6.6.2 Η συνάρτηση εύρεσης αλφαριθμητικού σε αλφαριθμητικό

Η συνάρτηση **strstr(str1,str2)** αναζητά μέσα στο αλφαριθμητικό **str1** το αλφαριθμητικό **str2**. Στην πρώτη εύρεση του **str2** η συνάρτηση επιστρέφει έναν δείκτη σε χαρακτήρα. Εάν δε βρεθεί το **str2**, επιστρέφεται η μηδενική διεύθυνση.

Η συνάρτηση **strstr()** δέχεται δύο ορίσματα: το πρώτο όρισμα είναι το όνομα του αλφαριθμητικού, στο οποίο θα γίνει η αναζήτηση και το δεύτερο όρισμα είναι το προς εύρεση αλφαριθμητικό. Η συνάρτηση **strstr()** ορίζεται στο αρχείο κεφαλίδας **string.h**.

Όπως και στην περίπτωση της συνάρτησης **strchr()**, ο δείκτης που επιστρέφει η συνάρτηση ουσιαστικά αντιστοιχεί στο τμήμα του αλφαριθμητικού που αρχίζει από τον πρώτο χαρακτήρα του **str2** και εκτείνεται έως το τέλος του **str1**.

6.6.2.1 Παράδειγμα

Στο πρόγραμμα που ακολουθεί, αναζητάται το αλφαριθμητικό **'angular'** μέσα στο αλφαριθμητικό **"tri-angular"**.

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

int main()
{
  char msg1[81]={"triangular"},msg2[]={"angular"};
  char *pfnd;
  printf( "\nmsg1: %s\n",msg1 );
  printf( "Address of string msg1: %d\n",msg1 );
  pfnd = strstr(msg1,msg2);
  printf("\npfnd: %s\n",pfnd);
  printf( "Address of string handled by pointer pfnd: %d\n",pfnd );

  return 0;
}
```

```
msg1: triangular
Address of string msg1: 2358752

pfnd: angular
Address of string handled by pointer pfnd: 2358755
```

Εικόνα 6.5 *Η έξοδος του προγράμματος του παραδείγματος 6.6.2.1*

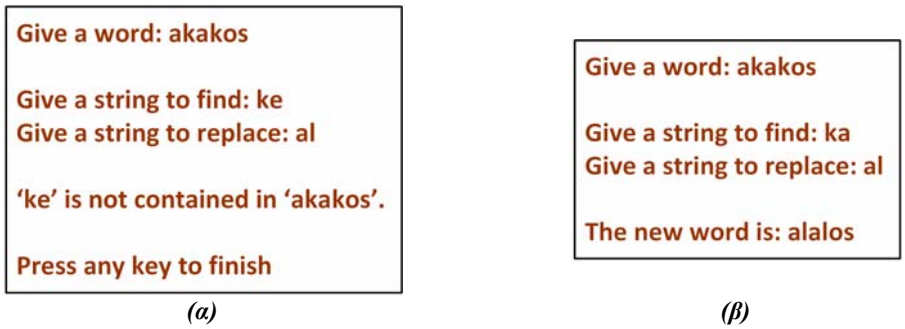
6.6.2.2 Παράδειγμα

Στο πρόγραμμα που ακολουθεί υλοποιείται η συνάρτηση **void replace(char *pword, char *poldString, char *pnewString)**, στην οποία οι δείκτες σε χαρακτήρα **pword**, **poldString** και **pnewString** μέσω της κλήσης κατ' αναφορά χειρίζονται τα αλφαριθμητικά **word**, **fnd**, **repl**, τα οποία δίνει ο χρήστης στη συνάρτηση **main()**. Σε κάθε εμφάνιση του αλφαριθμητικού **fnd** μέσα στο αλφαριθμητικό **word**, η συνάρτηση αντικαθιστά το **fnd** με το αλφαριθμητικό **repl**. Χάριν ευκολίας, το μήκος του **fnd** είναι πάντοτε ίδιο με αυτό του **repl**. Αρχικά, πρέπει να γίνει έλεγχος κατά πόσον το **fnd** υπάρχει μέσα στο **word**.

Πέραν της ανάγνωσης των αλφαριθμητικών, η συνάρτηση **main()** επιτελεί τα ακόλουθα:

- Καλεί τη συνάρτηση **replace(,)** με ορίσματα τις διευθύνσεις των αλφαριθμητικών **word**, **fnd**, **repl**.
- Μετά το πέρας της συνάρτησης **replace()** τυπώνεται στην οθόνη το νέο περιεχόμενο του αλφαριθμητικού **word**.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void replace(char *pword, char *poldString, char *pnewString);
int main()   {
  char word[21],fnd[11],repl[11];
  printf( "Give a word: " );            scanf( "%s",word);
  printf( "\nGive a string to find: ");  scanf( "%s",fnd );
  printf( "Give a string to replace: " );  scanf( "%s",repl );

  replace(word,fnd,repl);
  printf( "\nThe new word is: %s",word);
  return 0;
}
void replace(char *pword, char *poldString, char *pnewString)
{
  int i;
  char *temp;
  temp=strstr(pword,poldString);
  if (temp!=NULL)
  {
    do
    {
      for (i=0;i<strlen(poldString);i++)  temp[i]=pnewString[i];
      temp=temp+strlen(poldString);
      temp=strstr(temp,poldString);
    } while (temp!=NULL);
  }
  else
  {
    printf( "\n'%s' is not contained in '%s'.\n\nPress any key to
finish",poldString,pword );
    getchar();
    abort();
  }
}
```



(α) *(β)*

Εικόνα 6.5 *Η έξοδος του προγράμματος του παραδείγματος 6.6.2.2*

6.6.3 Υλοποίηση συναρτήσεων αλφαριθμητικών με χρήση δεικτών

Στα προγράμματα που ακολουθούν δίνονται τα πρωτότυπα των συναρτήσεων:
(α) **`strlen()`** (υπολογισμός του μήκους 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 *p1);
```

Το σώμα της συνάρτησης με δείκτες δίνεται παρακάτω:

```
int strlen(char *p1)
{
  char *p2=p1;
  while (*p1!='\0') p1++;
  return(p1-p2);
}
```

Ο δείκτης **`p1`** δείχνει στη διεύθυνση του αλφαριθμητικού (στον πρώτο χαρακτήρα). Στη γραμμή κώδικα **`char *p2=p1;`** ο δείκτης **`p2`** αποκτά το περιεχόμενο του **`p1`**, δείχνοντας και αυτός στον πρώτο χαρακτήρα. Ακολούθως, ο βρόχος **`while`** εκτελείται όσο ο **`p1`** δεν δείχνει στον μηδενικό χαρακτήρα: σε κάθε επανάληψη ο **`p1`** δείχνει στον επόμενο χαρακτήρα. Όταν ο βρόχος τερματίζεται, ο **`p1`** δείχνει στον μηδενικό χαρακτήρα, οπότε η εντολή **`p1-p2`** εκτελεί αφαίρεση δεικτών και δίνει τον αριθμό των στοιχείων μεταξύ των δύο δεικτών, όπερ σημαίνει 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 *p1, char *p2);
```

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

```
void strcpy(char *p1, char *p2)
{
  while ((*p1=*p2)!='\0')
  {
    p1++;
    p2++;
  }
}
```

Μέσα στη συνθήκη ελέγχου της επαναληπτικής πρότασης **`while`** πρώτα αντιγράφεται ο χαρακτήρας, στον οποίο δείχνει ο **`p2`** στη θέση που δείχνει ο **`p1`**, και ακολούθως ελέγχεται εάν ο δείκτης **`p2`** έφτασε να δείχνει στον μηδενικό χαρακτήρα. Κατά συνέπεια, εφόσον ο **`p2`** δείξει στον μηδενικό χαρακτήρα, αυτός πρώτα θα αντιγραφεί στη θέση που δείχνει ο **`p1`** και μετά η συνθήκη θα καταστεί ψευδής. Με αυτόν τον τρόπο ο **`p1`** θα αποκτήσει στο τέλος τον μηδενικό χαρακτήρα και η αντιγραφείσα σειρά χαρακτήρων θα αποτελεί ένα ολοκληρωμένο αλφαριθμητικό.

6.7 Ορίσματα της γραμμής εντολών

Κατ' αντιστοιχία με τις υπόλοιπες συναρτήσεις της γλώσσας C, η **`main()`** μπορεί να δεχτεί παραμέτρους, οι οποίες επιτρέπουν να δίνεται στο καλούμενο πρόγραμμα ένα σύνολο από εισόδους, οι οποίες ονομάζονται

ορίσματα της γραμμής εντολών (command line arguments). Ο μηχανισμός μεταβίβασης 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, στον οποίο συμπεριλαμβάνεται το όνομα του προγράμματος. Επομένως, η τιμή του **argc** είναι πάντοτε τουλάχιστον **1**.
- **argv** (*argument vector*): είναι δείκτης σε πίνακα δεικτών, που δείχνουν στα ορίσματα της γραμμής διαταγής. Τα ορίσματα αποθηκεύονται ως αλφαριθμητικά. Ο πίνακας, στον οποίο δείχνει ο **argv**, έχει ένα επιπλέον στοιχείο, το **argv[argc]**, το οποίο έχει τιμή τη μηδενική διεύθυνση, **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 στον χρόνο εκτέλεσης έχει ως αποτέλεσμα τη δέσμευση δύο θέσεων μνήμης για μελλοντική αποθήκευση δεικτών σε ακεραίους. Αντίθετα, η δεύτερη δήλωση ορίζει έναν δείκτη σε πίνακα δύο ακεραίων, τον οποίο όμως δεν δηλώνει και, κατά συνέπεια, δεν δεσμεύει τον απαιτούμενο χώρο.

6.7.1 Παράδειγμα

Στο πρόγραμμα που ακολουθεί, τυπώνονται τα ορίσματα της γραμμής διαταγής.

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

int main(int argc, char *argv[])
{
  int i;
  if (argc==1)
  {
     printf( "No arguments. Program aborted...\n" );
     exit(-1);
  }
  for (i=1;i<argc;i++)
     printf( "Argument no %d: %s",i,argv[i]\n" );

  return 0;
}
```

Έστω ότι το πρόγραμμα ονομάζεται **printArgs.c**. Όταν το καλούμε χωρίς ορίσματα, εκτελείται η πρόταση **if**, ενώ με την εντολή **printArgs first second third** το πρόγραμμα τυπώνει στην οθόνη:

```
Argument no 1: first
Argument no 2: second
Argument no 3: third
```

Το σώμα της **main()** έχει πρόσβαση στη μεταβλητή **argc**, η οποία λαμβάνει την τιμή **3**, και στον πίνακα δεικτών **argv**. Οι δείκτες που αποτελούν τα στοιχεία **argv[0]**, **argv[1]**, **argv[2]** δείχνουν στα

αλφαριθμητικά **"first"**, **"second"** και **"third"**, αντίστοιχα, ενώ το στοιχείο **argv[3]** έχει την τιμή **NULL**.

6.8 Δείκτες σε συναρτήσεις

Καθώς στη γλώσσα C οι συναρτήσεις έχουν τη δική τους διεύθυνση στη μνήμη, εκτός από τον παραδοσιακό τρόπο κλήσης με το όνομά τους μπορούν να κληθούν και μέσω δείκτη που θα «δείχνει» σε συνάρτηση (function poinrter). Επιπλέον, ο χειρισμός μίας συνάρτησης μέσω δείκτη δίνει τη δυνατότητα στη συνάρτηση να αποτελέσει παράμετρο άλλης συνάρτησης. Η λειτουργία αυτή ονομάζεται callback και χρησιμοποιείται κυρίως στην κατάστρωση κώδικα για συναρτήσεις βιβλιοθήκης.

Η δήλωση δείκτη σε συνάρτηση ακολουθεί τον εξής φορμαλισμό:

```
<επιστρεφόμενος τύπος δεδομένων> *<όνομα δείκτη>(παράμετροι);
```

Έστω οι συναρτήσεις **void func1(int x, char y)** και **double *func2(int x, char y)**. Οι δείκτες συναρτήσεων δηλώνονται αντίστοιχα ως:

```
void (*pfunc1)(int, int);
double *(*pfunc2)(int, char);
```

Παρατηρήσεις:

1. Ένας δείκτης σε συνάρτηση μπορεί να δείξει σε όλες τις συναρτήσεις που συμφωνούν με τη δήλωση του δείκτη, αρκεί να οι συναρτήσεις αυτές να έχουν τον ίδιο επιστρεφόμενο τύπο δεδομένων και την ίδια λίστα ορισμάτων.

2. Συνιστάται να γίνεται αρχικοποίηση των δεικτών σε συνάρτηση με την τιμή **NULL**.

3. Όπως το όνομα ενός πίνακα ταυτίζεται με τη διεύθυνση του πρώτου byte του πρώτου στοιχείου του, έτσι και το όνομα μίας συνάρτησης – χωρίς τις παρενθέσεις – παρέχει τη διεύθυνσή της.

6.8.1 Παράδειγμα

Στο ακόλουθο πρόγραμμα παρουσιάζεται ένα παράδειγμα χρήσης των δεικτών σε συναρτήσεις:

```
#include <stdio.h>

float funcPwr(float x, int n);

int main()
{
  /* δείκτης στη συνάρτηση funcPwr() */
  float (*pfuncPwr)(float,int)=NULL;
  printf( "Address of function funcPwr=%d\n",funcPwr );
  pfuncPwr=&funcPwr;
  printf( "Pointer pfuncPwr points to address %d\n",pfuncPwr );
  printf(     "\nDirect     function     call:     %.2f^%d     =
%.2f\n",2.0,6,funcPwr(2.0,6) );
  printf("  Function  call  via  function  pointer:  %.2f^%d  =
%.2f\n",2.0,6,pfuncPwr(2.0,6) );

  return 0;
}

float funcPwr(float x, int n)
{
  int i;
  float prod=1.0;
```

```
    if (n==0) return 1.0;
    else
    {
       for (i=1;i<=n;i++)   prod=prod*x;
       return prod;
    }
}
```

```
Address of function funcPwr=4199946
Pointer pfuncPwr point to Address 4199946

Direct function call: 2.00^6 = 64.00
Function call via function pointer: 2.00^6 = 64.00
```

Εικόνα 6.6 *Η έξοδος του προγράμματος του παραδείγματος 6.8.1*

Μελετώντας τα αποτελέσματα από την εκτέλεση του προγράμματος συμπεραίνεται ότι ο δείκτης **pfuncPwr** έχει ως περιεχόμενο τη διεύθυνση της συνάρτησης **funcPwr**. Επίσης, συνάγεται ότι τόσο η άμεση κλήση της συνάρτησης όσο και η κλήση μέσω του δείκτη **pfuncPwr** οδηγούν στη λειτουργία της συνάρτησης με τον ίδιο ακριβώς τρόπο.

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποιος από τους ακόλουθους τρόπους αρχικοποίησης δεικτών είναι λανθασμένος;

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

(β)
```
int *pnum;
int count;
pnum=&count;
```

(γ)
```
int numArray[5]={1,2,3,4,5};
int *pint;
pint=numArray[0];
```

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

(2) Ποιο είναι το αποτέλεσμα της εκτέλεσης του ακόλουθου προγράμματος;

```
#include <stdio.h>
int *func(int *p)
{
   *p=42;
   return(p);
};
int main()
{
   int p=13;
```

```
   int *z;
   printf( "p=%d\n",p );
   z=func(&p);
   printf( "(*z)=%d p=%d",*z,p );

   return 0;
}
```

(α) `p=13`
`(*z)=13 p=13`

(β) `p=13`
`(*z)=42 p=42`

(γ) Θα εμφανιστούν σφάλματα κατά τη μεταγλώττιση.

(δ) `p=13`
`(*z)=13 p=42`

(3) Ποιο είναι το αποτέλεσμα της εκτέλεσης του ακόλουθου προγράμματος;

```
int f(int *i)
{
   return((*i)+1);
}
int  main()
{
   int n=10;
   printf( "%d",f(&n) );

   return 0 ;
}
```

(α) 10

(β) Θα εμφανιστούν σφάλματα κατά τη μεταγλώττιση/αποσφαλμάτωση.

(γ) 9

(δ) 11

(4) Στις συναρτήσεις με τύπο επιστροφής δείκτη, ποια από τις ακόλουθες συμβάσεις για την επιστρεφόμενη τιμή είναι σωστή;

(α) Η επιστρεφόμενη τιμή μπορεί να είναι διεύθυνση οιασδήποτε μεταβλητής της συνάρτησης.

(β) Η C δεν υποστηρίζει συναρτήσεις με τύπο επιστροφής δείκτη.

(γ) Η επιστρεφόμενη τιμή μπορεί να είναι διεύθυνση ακέραιας μεταβλητής της συνάρτησης.

(δ) Η επιστρεφόμενη τιμή μπορεί να είναι διεύθυνση static μεταβλητής της συνάρτησης, οιουδήποτε τύπου.

(5) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;

(α) Ένας δείκτης μπορεί να περιέχει μόνο τη διεύθυνση μίας θέσης μνήμης.

(β) Ο τελεστής ***** χρησιμοποιείται για να έχουμε πρόσβαση σε μία θέση μνήμης μέσω ενός δείκτη, ο οποίος περιέχει τη διεύθυνσή της.

(γ) Εάν εφαρμόσουμε τον τελεστή **++** σε μία μεταβλητή δείκτη, αυτή αυξάνεται κατά 1.

(δ) Το μέγεθος μίας μεταβλητής δείκτη εξαρτάται από το σύστημα ,στο οποίο εκτελείται το πρόγραμμά μας.

Ασκήσεις

Άσκηση 1

Να γραφεί πρόγραμμα, στο οποίο θα λαμβάνεται από το πληκτρολόγιο ένας ακέραιος αριθμός και θα καλείται με κλήση κατ' αναφορά η συνάρτηση `void cube(int *pnumber)`, η οποία θα υπολογίζει την τρίτη δύναμη του αναγνωσθέντος αριθμού.

Άσκηση 2

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

- **Θα λαμβάνεται από το πληκτρολόγιο ένα αλφαριθμητικό.**
- **Θα καλείται με κλήση κατ' αναφορά η συνάρτηση `void rev(int *pstring)`, η οποία θα τοποθετεί τους χαρακτήρες του αλφαριθμητικού σε αντίστροφη σειρά.**
- **Θα καλείται με κλήση κατ' αναφορά η συνάρτηση `void conv(int *pstring)`, η οποία θα μετατρέπει τους πεζούς χαρακτήρες του αντιστραφέντος αλφαριθμητικού σε κεφαλαίους και τανάπαλιν.**
- **Το προκύπτον αλφαριθμητικό θα εμφανίζεται στην οθόνη.**

Άσκηση 3

Να καταστρωθεί πρόγραμμα, το οποίο θα λαμβάνει από το πληκτρολόγιο δύο αλφαριθμητικά και θα τα μεταβιβάζει με κλήση κατ' αναφορά στη συνάρτηση `str_index(char *s, char *t)`. Οι δείκτες σε χαρακτήρα θα χειρίζονται τα δύο αλφαριθμητικά. Ακολούθως, μέσω του κώδικα της συνάρτησης θα υπολογίζεται η θέση της δεξιότερης εμφάνισης του αλφαριθμητικού `t` μέσα στο `s`, η οποία και θα επιστρέφει στη `main()`. Εάν το `t` δεν υπάρχει μέσα στο `s`, θα επιστρέφεται το 0. Στη `main()` θα εμφανίζεται κατάλληλο μήνυμα για τη θέση στην οποία εμφανίζεται το `t` ή για τη μη εμφάνισή του. Για την ορθή λειτουργία της συνάρτησης `str_index` απαιτείται χρήση της συνάρτησης `strstr()`, η οποία ορίζεται στο αρχείο κεφαλίδας `string.h`.

Άσκηση 4

Να περιγραφεί αναλυτικά η λειτουργία των ακόλουθων προγραμμάτων και να δοθούν τα αποτελέσματά τους.

(α)

```
#include <stdio.h>

int func(int *i)
{
  static int j=1;
  j=j+(*i);
  return((*i)+j);
}
int main()
{
  int n;
  for (n=0;n<8;n=n+3)
    printf( "%d\n",func(&n) );
  return 0;
}
```

(β)

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

char *func(char *i)
{
  return(i+1);
}
int main()
{
  char *fnd,n[10]="Fourteen";
  fnd=func(&n[2]);
  printf( "%s",fnd );
  return 0;
}
```

Άσκηση 5

Να γραφεί μία συνάρτηση `int add(int *pin, int number)`, η οποία θα δέχεται ως ορίσματα: *(α)* τη διεύθυνση του πρώτου στοιχείου ενός μονοδιάστατου πίνακα ακεραίων και *(β)* το πλήθος των στοιχείων του. Η συνάρτηση θα επιστρέφει το άθροισμα των τετραγώνων των στοιχείων του πίνακα.

Άσκηση 6

Να τροποποιηθεί το πρόγραμμα του παραδείγματος 6.7.1, ώστε τα ορίσματα της γραμμής εντολών να εμφανίζονται στην οθόνη με αντρίστροφη σειρά.

Άσκηση 7

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

(α) **Θα δημιουργείται δισδιάστατος πίνακα αριθμών κινητής υποδιαστολής `arr`, διαστάσεων 4x5.**

(β) **Θα καλείται η συνάρτηση `void read_data(float **parr, int m, int n)`, η οποία θα αποδίδει τιμές στον `arr` (έμμεσα) από το πληκτρολόγιο.**

(γ) **Θα καλούνται οι συναρτήσεις `float maxRow(float *prow, int n)`, `float minRow(float *prow, int n)`, `float meanRow(float *prow, int n)`, οι οποίες για κάθε γραμμή του πίνακα `arr` θα επιστρέφουν στη `main()` τη μέγιστη, την ελάχιστη και τη μέση τιμή των στοιχείων της γραμμής, αντίστοιχα. Θα εμφανίζονται στην οθόνη τόσο ο πίνακας `arr` όσο και οι στατιστικές τιμές για κάθε γραμμή.**

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Kernighan, B. & Ritchie, D. (1990), *Η γλώσσα προγραμματισμού C*, Εκδόσεις Κλειδάριθμος.

Kernighan, B. & Pike, R. (1999), *The Practice of Programming*, Addison-Wesley.

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

Prata, S. (2014), *C Primer Plus*, 6th ed., Addison-Wesley.

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

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

7. Δυναμική διαχείριση μνήμης

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

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

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

7.1 Η έννοια της δυναμικής διαχείρισης μνήμης

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

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

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

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

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

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

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

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

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

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

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

7.2 Οι συναρτήσεις malloc, calloc και free

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Παρατηρήσεις:

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

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

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

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

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

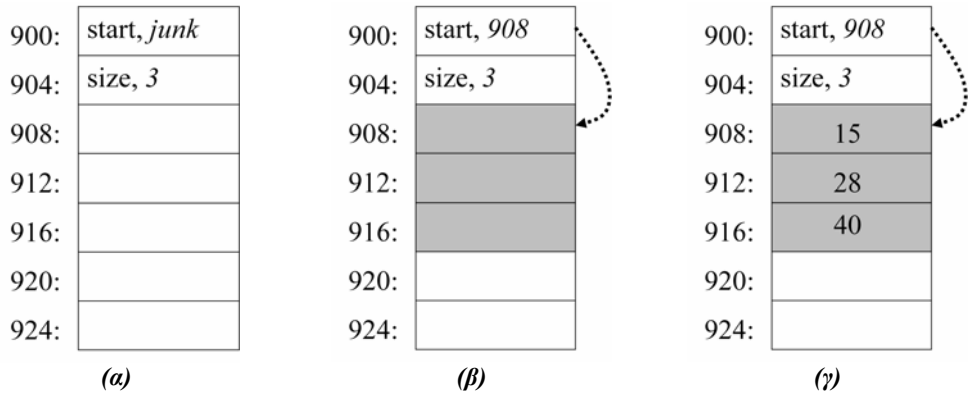
```
pmessage=(char *)malloc(20*sizeof(char);
. . . . . .
pmsg=pmessage;/* Πλέον και οι δύο δείκτες δείχνουν στην ίδια θέση */
free(pmsg);
free(pmessage); /* ΛΑΘΟΣ: Το μπλοκ μνήμης έχει ήδη απελευθερωθεί! */
```

7.2.1 Παράδειγμα

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

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

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

  return 0;
}
```

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



Σχήμα 7.1 *Χάρτης μνήμης του παραδείγματος 7.2.1*

7.3 Η συνάρτηση realloc

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

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

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

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

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

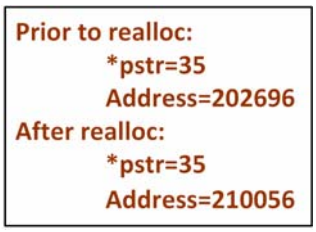
με την εκχώρηση της διεύθυνσης του πρώτου byte της δεσμευθείσας μνήμης στον δείκτη **ptr**, τύπου **double**, γίνεται αυτόματη μετατροπή τύπου και ο **ptr** μπορεί να διαχειριστεί τα **8x10=80** bytes της δεσμευθείσας μνήμης.

7.3.1 Παράδειγμα

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

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

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

  return 0;
```

```
}
```



Εικόνα 7.2 *Η έξοδος του προγράμματος του παραδείγματος 7.3.1*

7.4 Μονοδιάστατοι δυναμικοί πίνακες

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

1. Δηλώνεται ένας δείκτης που θα διαχειριστεί τον πίνακα.

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

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

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

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

```
double *ptr;
```

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

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

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

Η αποδέσμευση του δεσμευμένου χώρου μνήμης γίνεται ως εξής:

```
free(ptr);
```

<u>Παρατηρήσεις:</u>

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

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

7.5 Πίνακας δεικτών για τη διαχείριση αλφαριθμητικών

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

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

Η πρόταση

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

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

7.5.1 Παράδειγμα

Θεωρούμε το ακόλουθο πρόγραμμα.

```
#include<stdio.h>

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

  return 0;
}
```

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

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

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

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

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

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

Εικόνα 7.3 *Η έξοδος του προγράμματος του παραδείγματος 7.5.1*

7.6 Δείκτης σε δείκτες για τη διαχείριση πολυδιάστατων πινάκων δεδομένων

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

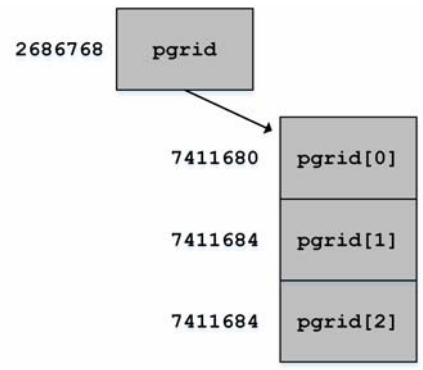
Η έμμεση αναφορά μπορεί να λάβει ένθεση οιουδήποτε βάθους (δείκτης σε δείκτη σε δείκτη κ.λπ.), ωστόσο θα πρέπει να αποφεύγονται οι υπερβολές, γιατί ο κώδικας αφενός μεν θα γίνει δυσανάγνωστος, αφετέρου θα είναι επιρρεπής σε σφάλματα.

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

```
#include <stdio.h>

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

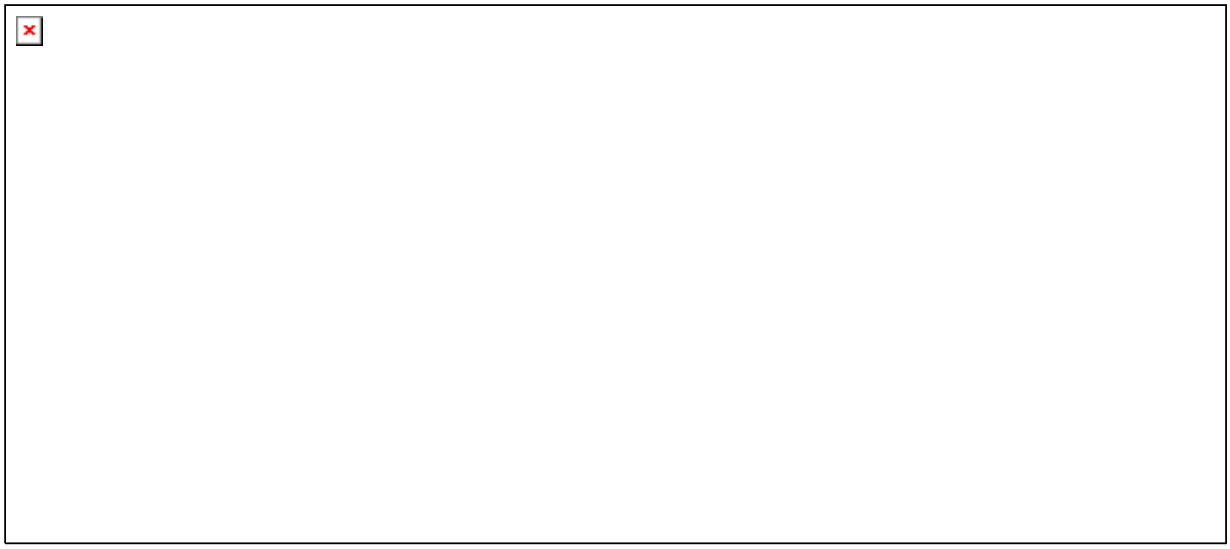
```
    return 0;
}
```

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



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

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



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

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

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

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

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

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

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

```
address of pmatr=2686780

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

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

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

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

Εικόνα 7.4 *Η έξοδος του προγράμματος του παραδείγματος 7.6*

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

7.6.1 Παράδειγμα

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

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

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

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

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

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

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

7.7.1 Παράδειγμα

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

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

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

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

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

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

  return 0;
}

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

  return(parr);
}

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

Εικόνα 7.5 *Η έξοδος του προγράμματος του παραδείγματος 7.7.1*

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

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

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

7.7.2 Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

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

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

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

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

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

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

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

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

Δίνεται ότι:

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

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

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

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

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

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

```
  gets(all[1]);

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

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

  return 0;
}

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

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

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

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

    j=0;
    do
```

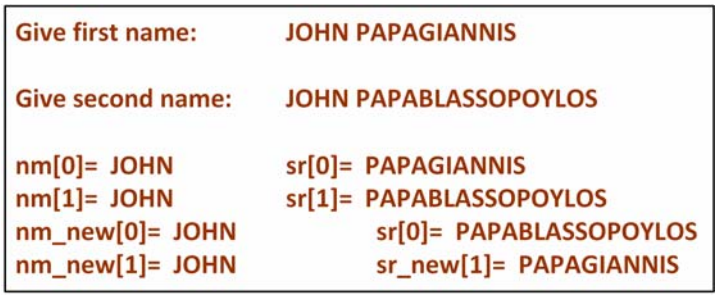
```
    {
      psr[i][j]=pall[i][strlen(pnm[i])+j];
      j++;
    } while (pall[i][j]!='\0');
  }
}

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

```
Give first name:        JOAN JAMESON

Give second name:       ANDREW DOW

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

(α)

```
Give first name:        JOHN HATZISAVVAS

Give second name:       JOHN HATZIS

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

(β)



(γ)

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

7.8. Παράδειγμα ανάπτυξης προγράμματος

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

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

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

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

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

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

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

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

i. ανάγνωση του μεγέθους του πίνακα,

ii. δημιουργία πίνακα,

iii. ανάγνωση των στοιχείων του πίνακα,

iv. εκτύπωση των στοιχείων του πίνακα,

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

vi. υπολογισμός του ίχνους του πίνακα,

vii. αντιμετάθεση δύο στηλών του πίνακα,

viii. αντιμετάθεση δύο γραμμών του πίνακα.

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

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

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

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

  return size;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  return trc;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
  free_2(pArr,size);

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

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

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

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

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

```
}
```

```
Give the size of the array:     3

A[1][1]:        23.1

A[1][2]:        -32

A[1][3]:        5.7

A[2][1]:        1.15

A[2][2]:        -0.02

A[2][3]:        0.17

A[3][1]:        3.41

A[3][2]:        4.2

A[3][3]:        -1

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

Trace(A)=22.080000


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

Εικόνα 7.4 *Η έξοδος του προγράμματος*

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

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;

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

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

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

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

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

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

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

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

```
    return 0;
}
```

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

Ασκήσεις

Άσκηση 1

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

Άσκηση 2

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

Άσκηση 3

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

Άσκηση 4

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

Άσκηση 5

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

Άσκηση 6

Να γραφεί πρόγραμμα το οποίο θα επιτελεί τα ακόλουθα:

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

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

Βιβλιογραφία κεφαλαίου

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

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

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

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

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

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

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

8. Απαριθμητικοί τύποι δεδομένων – Δομές – Ενώσεις

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια των τύπων δεδομένων που ορίζονται από τον χρήστη. Αρχικά, μελετώνται οι απαριθμητικοί τύποι και αναλύονται τα χαρακτηριστικά και οι περιορισμοί τους. Ακολούθως, παρουσιάζεται η δομή δεδομένων και δίνονται ο ορισμός και ο τρόπος λειτουργίας της. Περιγράφονται οι ένθετες δομές, οι δομές ως ορίσματα συναρτήσεων και ως επιστρεφόμενες τιμές συναρτήσεων, οι δομές με μέλη δείκτες καθώς και οι δείκτες σε δομές. Στην επόμενη ενότητα αναλύεται η ένωση ως τύπος δεδομένων και παρουσιάζεται ο τρόπος λειτουργίας της. Το κεφάλαιο ολοκληρώνεται με ένα εκτενές παράδειγμα ανάπτυξης προγράμματος.

Λέξεις κλειδιά
απαριθμητικοί τύποι (enum) – typedef – δομή (struct) – ένθεση δομών – πίνακας δομών – δείκτης σε δομή – ένωση (union) – τελεστής τελεία – τελεστής βέλος.

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

8.1 Απαριθμητικοί τύποι δεδομένων

Η γλώσσα C, πέραν των ενσωματωμένων τύπων δεδομένων (**char**, **int**, **float**, **double**), παρέχει τη δυνατότητα στον χρήστη να δημιουργήσει τους δικούς του τύπους, οι οποίοι είναι ενεργοί αποκλειστικά μέσα στο πρόγραμμα που ορίζονται. 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 είναι ένα σύνολο από σταθερές, στις οποίες έχουν δοθεί συμβολικά ονόματα. Η σημασία των απαριθμητικών τύπων είναι διττή: αφενός διευκολύνουν την κατανόηση του προγράμματος, αφετέρου επιτρέπουν στον μεταγλωττιστή να ελέγχει τους τύπους, ώστε να μην προκαλούνται λογικά λάθη. Για παράδειγμα, εάν χρησιμοποιηθεί μία ακέραια μεταβλητή **days**, για να επεξεργαστεί 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 δεν θα εντοπίσει το λάθος, καθώς υπάρχει λογικό και όχι προγραμματιστικό σφάλμα. Για τον λόγο αυτόν μπορεί να οριστεί ένας νέος τύπος δεδομένων με το όνομα **weekDays**, με πεδίο τιμών επτά συμβολικά ονόματα, καθένα από τα οποία αντιστοιχεί σε μία ημέρα της εβδομάδας:

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

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

```
enum weekDdays days;
```

Παρατηρήσεις:

1. Εσωτερικά, ο χειρισμός των τύπων δεδομένων **enum** γίνεται σαν να είναι ακέραιοι (έτσι μπορούν να εκετελεστούν 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 weekDdays { 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.

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

```
enum weekDays { 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 οριστούν 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 οριστούν boolean μεταβλητές.

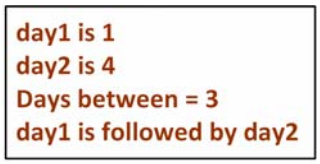
Εναλλακτικά, τα παραπάνω επιτελούνται με χρήση της πρότασης **#define** του προεπεξεργαστή:

```
#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.

8.1.1 Παράδειγμα

```
#include <stdio.h>

/* Προσδιορισμός τύπου enum */
enum weekDays { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

int main()
{
  weekDays day1,day2;  /* ορισμός μεταβλητών */
  day1=Mon; /* Απόδοση τιμής. Δεν περικλείεται σε εισαγωγικά. */
  day2=Thu;
  int diff=day2-day1;  /* Ακολουθείται η αριθμητική ακεραίων */
  printf( "day1 is %d\n",day1 );
  printf( "day2 is %d\n",day2 );
  printf( "Days between =%d\n",diff );
  /* Στους απαριθμητικούς τύπους γίνονται  συγκρίσεις */
  if (day1<day2)
    printf( "day1 is followed by day2\n" );

  return 0;
}
```



Εικόνα 8.1 *Η έξοδος του προγράμματος του παραδείγματος 8.1.1*

8.2 Η λέξη κλειδί typedef

Η γλώσσα 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  realNumber;
```

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

```
typedef int (* PTR) (char *, char);
```

δημιουργεί τον τύπο **PTR**, ο οποίος είναι δείκτης σε συνάρτηση που επιστρέφει ακέραια τιμή και έχει ως ορίσματα έναν δείκτη σε χαρακτήρα και έναν χαρακτήρα.

Η χρήση συνωνύμων διευκολύνει τον προγραμματισμό και επιτρέπει την επέκταση της γλώσσας. Επιπλέον, ενισχύει τη φορητότητα των προγραμμάτων, καθώς εάν χρησιμοποιηθεί η λέξη κλειδί **typedef** για τους τύπους δεδομένων που εξαρτώνται από το μηχάνημα (π.χ. οι ακέραιες ποσότητες), το μόνο που θα χρειαστεί για να μεταφερθεί το πρόγραμμα σε μηχάνημα διαφορετικής αρχιτεκτονικής είναι η αλλαγή των δηλώσεων **typedef**.

8.3 Ορισμός του τύπου δεδομένου δομής – δήλωση μεταβλητών τύπου δομής

Η δομή αποτελεί έναν *συναθροιστικό* (aggregate) τύπο δεδομένων, οριζόμενο από τον 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 οριστεί 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 οριστεί πριν τον ορισμό της δομής.
- Πίνακες.
- Άλλες δομές, οι οποίες θα πρέπει να έχουν οριστεί πριν τον ορισμό της δομής.

Ο ορισμός της δομής έχει την ακόλουθη μορφή:

```
struct <όνομα δομής>
{
     <τύπος δεδομένων> <όνομα 1ου μέλους>;
     <τύπος δεδομένων> <όνομα 2ου μέλους>;
```

```
    . . . . . .
    <τύπος δεδομένων> <όνομα τελευταίου μέλους>;
};
```

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**, η οποία περιέχει τέσσερα μέλη:

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

ή εναλλακτικά:

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

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

```
struct addrT address1,address2;
```

ορίζει ότι οι **address1**, **address2** είναι μεταβλητές τύπου **addrT**. Η δήλωση των μεταβλητών μπορεί να συνδυαστεί 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 zipCode;
} address1, address2;
```

Με παρόμοιο τρόπο δηλώνεται κι 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. 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 τον χρήστη.

2. 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**.

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

4. Ο ορισμός μίας δομής δεν δεσμεύει χώρο στη μνήμη για μεταβλητές. Απλά δημιουργεί έναν νέο τύπο.

5. 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 zipCode;
} address1;
```

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

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

6. 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.λπ.).

7. Οι λειτουργίες που μπορούν να εκτελεστούν 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 της **address1** αποκτούν τις τιμές των αντίστοιχων μελών της **address2**.
- Η απόδοση αρχικών τιμών σε μία μεταβλητή τύπου δομής.
- Η αναφορά στα μέλη μίας μεταβλητής τύπου δομής.
- Η διεύθυνση μίας μεταβλητής τύπου δομής μπορεί να ανατεθεί σε έναν δείκτη (pointer).
- 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**.

8.3.1 Παράδειγμα

Μία επιχείρηση πώλησης αυτοκινήτων διαθέτει αυτοκίνητα με τα χαρακτηριστικά που παρατίθενται στον **Πίνακα 8.1**:

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

Πίνακας 8.1

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 απλής ακρίβειας για την τιμή του κάθε μοντέλου
avail: ακέραιος για τον αριθμό των διαθέσιμων τεμαχίων κάθε μοντέλου

Για το πρώτο μέλος θα δημιουργηθεί ο απαριθμητικός τύπος **make**, ο ορισμός του οποίου θα προηγείται του ορισμού της δομής:

```
enum carmakeT { Mercedes, BMW, Audi };
```

Κατά συνέπεια, ο κώδικας ορισμού της δομής και της 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;
};

int 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:

1. Είτε μαζί με τον ορισμό και τη δήλωση:

```
struct addrT
{
  char name[40],street[40],city[30];
  unsigned int zipCode;
} address1={"Demis Pappas","Rodou 23","Serres",62124},
address2={"John Doe","Limnou 32","Serres",62124};
```

2. Είτε με τη δήλωση:

```
struct addrT address1={"Demis Pappas","Rodou 23","Serres",62124};
       address2={"John Doe","Limnou 32","Serres",62124};
```

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 τον 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",62124 },
  {"John Doe","Limnou 32","Serres",62124},
  {"Eleni Manta","Skirou 12","Serres",62124}
};
```

8.5 Αναφορά στα μέλη δομής

Η προσπέλαση των μελών μίας δομής γίνεται με χρήση δύο τελεστών: του **τελεστή μέλους δομής** (`.`) ή **τελεστή τελείας** (structure member operator) και του **τελεστή δείκτη δομής** (`->`) ή **τελεστή βέλους** ή **έμμεσης προσπέλασης** ή **δείκτη δομής** (structure pointer operator, arrow operator). Η χρήση του τελεστή βέλους περιγράφεται στην §8.8.

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 `address[9].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**. *Δεν θα πρέπει να συγχέεται ο τύπος δεδομένων που ορίστηκε με τις μεταβλητές τέτοιου τύπου.*

8.6 Ένθεση δομών

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

```
struct personT
{
  char name[40];
  char address[40];
  char tel[15];
  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 monthName[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 στον μήνα πρόσληψης, έχουν τη μορφή `bemp.birthdate.year` και `bemp.hiredate.month_name[2]` αντίστοιχα.

8.6.1 Παράδειγμα

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

```
#include <stdio.h>

struct addressT
{
  char name[40],street[40],city[40];
  int number,zipCode;
};   /* 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 */

int main()
{
  struct addressT addr1={"John Doe","Telou Agra",10,62124,"Serres"};

  struct addressT addr[10]={
    {"Ken Thomson","Rodou",23,61124,"Serres"},
    {"Brian Kernighan","Dilou",26,62124,"Serres"},
  };

  struct personT p={
    {"Brian Kernighan","Dilou",26,62124,"Serres"},
    {28,1,79},
  };

  printf( "\n\tstruct address\n" );
  printf( "%s\n%s %d\n%d\n%s\n",addr1.name,addr1.street,
  addr1.number,addr1.zipCode,addr1.city );
  printf( "\n\tstruct person\n" );
  printf( "%s\n%s %d\n%d\n%s\n",p.addr.name,p.addr.street,
  p.addr.number,p.addr.zipCode,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] );
}
```

Εικόνα 8.2 *Η έξοδος του προγράμματος του παραδείγματος 8.6.1*

Αρχικά, ορίζεται ο τύπος δεδομένων **`struct addressT`**. Η νέα δομή περιλαμβάνει τα μέλη **`name`**, **`street`**, **`number`**, **`zipCode`** 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.

Στις επόμενες γραμμές κώδικα 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 **`personT`**. Θα πρέπει να προσεχθεί ότι οι αρχικοποιήσεις κάθε μέλους της δομής περικλείονται σε άγκιστρα.

Στη συνέχεια της **`main()`** παρουσιάζονται προτάσεις εκτύπωσης. Στην πρόταση

```
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 πρόταση

```
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 πρόταση

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

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

8.6.2 Παράδειγμα

Για να δημιουργηθεί μία προσωπική ατζέντα, ορίζεται ο πίνακας **`directory[40]`**. Κάθε στοιχείο του πίνακα 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 το fax του καταγεγραμμένου.

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

Η εγγραφή και η εκτύπωση των στοιχείων του πίνακα **directory** θα γίνεται μέσα από τη συνάρτηση **main()**.

Παρατηρήσεις:

1. Σύμφωνα με τις ανωτέρω προδιαγραφές, 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 streetName[40],city[40];
  int streetNumber,zipCode;
};
```

3. Είναι προτιμητέο οι τηλεφωνικοί αριθμοί να δηλώνονται ως αλφαριθμητικά, γιατί αποτελούν 10-ψήφιους ή 14-ψήφιους ακέραιους (με το πρόθεμα της χώρας **0030**) ή αλφαριθμητικά (με το πρόθεμα της χώρας **+30**). Κατά συνέπεια, ο τύπος **teleT** μπορεί να έχει τη μορφή:

```
struct teleT
{
  char wrNo[15];       /* σταθερό τηλέφωνο */
  char cellNo[15];     /* κινητό τηλέφωνο */
  char faxNo[15];
} ;
```

Ο συνολικός κώδικας είναι ο ακόλουθος:

```
#include <stdio.h>

struct addressT
{
  char streetName[40],city[40];
  int streetNumber;
  int zipCode;
};

struct idT
{
  char name[40],surname[40];
```

```
   struct addressT addr;
};

struct teleT
{
   char wrNo[15],cellNo[15],faxNo[15];
} ;

struct emT
{
   char emWork[40];
   char emHome[40];
} ;

struct personT
{
   struct idT ident;
   struct teleT tel;
   struct emT email;
};

int main()
{
   struct 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.streetName );
      printf( "\n\tStreet number: ");
      scanf( "%d",&directory[i].ident.addr.streetNnumber );
      printf( "\n\tCity: " );
      scanf( "%s",directory[i].ident.addr.city );
      printf( "\n\tZip code: " );
      scanf( "%d",&directory[i].ident.addr.zipCode );
      printf( "\n\tTelephone: " );
      scanf( "%s",directory[i].tel.wrNo );
      printf( "\n\tCell telephone: ");
      scanf( "%s",directory[i].tel.cellNo );
      printf( "\n\tFax: ");
      scanf( "%s",directory[i].tel.faxNo );
      printf( "\n\tE-mail (work): ");
      scanf( "%s",directory[i].email.emWork );
      printf( "\n\tE-mail (home): " );
      scanf( "%s",directory[i].email.emHome );
   }

   return 0;
}
```

Εικόνα 8.3 *Η έξοδος του προγράμματος του παραδείγματος 8.6.2*

8.7 Συναρτήσεις με ορίσματα και επιστρεφόμενη τιμή τύπου δομής

Οι μεταβλητές τύπου δομής μπορούν να χρησιμοποιηθούν ως ορίσματα συναρτήσεων όπως και οι μεταβλητές των προκαθορισμένων τύπων (**float**, **int** κ.ο.κ.). Επιπλέον, οι συναρτήσεις μπορούν να επιστρέφουν τύπους δομής. Στο πρόγραμμα που ακολουθεί παρουσιάζονται συναρτήσεις που δέχονται και επιστρέφουν δομές.

Ο κώδικας υλοποιεί έναν αθροιστή μεγεθών, που είναι εκφρασμένα σε πόδια και ίντσες. Με βάση την αγγλοσαξονική μονάδα μέτρησης, εάν ο αριθμός των ιντσών υπερβαίνει το **12**, συμπληρώνεται ένα πόδι, επομένως πρέπει να προστεθεί η μονάδα στον αριθμό των ποδιών και να αφαιρεθεί το **12** από τον αριθμό των ιντσών.

```
#include <stdio.h>

struct DistT
{
  int feet;
  int inches;
};

DistT addDistances(struct DistT dist1, struct DistT dist2);
void displayDistance(struct DistT dist);

int main()
{
```

```
  struct DistT 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=addDistances(d1,d2);
  displayDistance(d1);
  printf( " + " );
  displayDistance(d2);
  printf( " = " );
  displayDistance(d3);

  return 0;
}

void displayDistance(struct DistT dist)
{
  printf( "%d'%d'' ",dist.feet,dist.inches );
}

DistT addDistances(struct DistT dist1, struct DistT dist2)
{
  struct DistT dist3;
  dist3.inches=dist1.inches+dist2.inches;
  dist3.feet=0;
  if (dist3.inches>=12)
  {
     dist3.inches=dist3.inches-12;
     dist3.feet++;
  }
  dist3.feet=dist3.feet+dist1.feet+dist2.feet;

  return(dist3);
}
```

```
Give the first number of feet: 5

Give the first number of inches: 4

Give the second number of feet: 3

Give the second number of inches: 9
5'4'' + 3'9'' = 9'1''
```

Εικόνα 8.4 *Η έξοδος του προγράμματος*

Αρχικά, ορίζεται η δομή **DistT** με μέλη τις ακέραιες μεταβλητές **feet** και **inches**. Ορίζεται η συνάρτηση **displayDistance**, η οποία δέχεται ως όρισμα (κλήση κατ' αξία) τις τιμές από τον τύπο δεδομένων **DistT**. Κατά την κλήση μίας συνάρτησης με ορίσματα τύπου δομής η αντιστοίχιση των μελών στην τοπική μεταβλητή γίνεται αυτόματα, ανεξάρτητα από την πολυπλοκότητα της εκάστοτε δομής.

Η συνάρτηση **addDistances**, η οποία δέχεται δύο μήκη και τα αθροίζει, έχει επιστρεφόμενη τιμή τύπου δομής **DistT**. Όπως και στην περίπτωση των ορισμάτων, έτσι και οι επιστρεφόμενη τιμή των μελών της δομής ανατίθεται αυτόματα στα αντίστοιχα μέλη της τοπικής μεταβλητής της καλούσας συνάρτησης, ανεξάρτητα από την πολυπλοκότητα της εκάστοτε δομής. Δηλαδή, η τιμή **dist3.feet** ανατίθεται στο μέλος **d3.feet** της μεταβλητής **d3** που βρίσκεται στη **main()**, ενώ η τιμή **dist3.inches** ανατίθεται στο μέλος **d3.inches** της μεταβλητής **d3**.

8.8 Δείκτες και δομές

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

```
struct addressT addr;
```

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

```
stuct addressT *paddr;
paddr=&addr;
```

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

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

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

τυπώνει το μέλος **name** της δομής **addr**, ενώ η πρόταση

```
paddr->zipCode=62124;
```

αναθέτει το **62124** στο μέλος **zipCode** της δομής **addr**.

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

8.8.1 Δείκτες εντός δομών

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

```
struct structTypeT
{
  int *point1;
  char *point2;
  float var3;
};

int main()
{
  struct structTypeT 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 */
      . . . . . .
  return 0;
}
```

8.8.2 Παράδειγμα

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

```
#include<stdio.h>

typedef struct vect /* Ορισμός της δομής vect */
{
  float x,y;
} vector; /* Η λέξη vector γίνεται συνώνυμη του vect */

void prvect(char d, vector v);
void scanvect(vector *p);
float inprodr(vector *p, vector *q);
vector addvectr(vector *p, vector *q);
/*-------------------------------------*/
int 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);
  return 0;
}
/*-------------------------------------*/
/* Εκτύπωση διανύσματος, κλήση κατ' αξία */
void prvect(char d, vector v)
{
  printf( "Vector %c is  ",d );
  printf( "(%.2f,%.2f)\n",v.x,v.y );
}
/*-------------------------------------*/
/* Ανάγνωση διανύσματος, κλήση κατ' αναφορά */
void scanvect( vector *p)
{
  printf( "Give the x co-ordinate: " );
  scanf( "%f",&p->x );
  printf( "Give the y co-ordinate: " );
  scanf( "%f",&p->y );
}
/*-------------------------------------*/
/* Εσωτερικό γινόμενο διανυσμάτων, κλήση κατ' αναφορά και
```

```
   επιστρεφόμενη τιμή προκαθορισμένου τύπου */
float inprodr(vector *p, vector *q)
{
   return((*p).x*(*q).x+(p->y)*(q->y));
}
/*-----------------------------------*/
/* Άθροισμα διανυσμάτων, κλήση κατ' αναφορά και επιστρεφόμενη τιμή
   τύπου δομής */
vector addvectr(vector *p, vector *q)
{
   vector sum;
   sum.x=(p->x)+(q->x);
   sum.y=(p->y)+(q->y);
   return(sum);
}
```

```
Give the x co-ordinate: -2.2
Give the y co-ordinate: 0.6
Vector a is  (-2.20,0.60)
Give the x co-ordinate: 0
Give the y co-ordinate: 1
Vector b is  (0.00,1.00)
        The inner product of a and b is: 0.60
Vector c is  (-2.20,1.60)
```

Εικόνα 8.5 *Η έξοδος του προγράμματος του παραδείγματος 8.8.2*

8.9 Ενώσεις

Η ένωση (union) είναι ένας χώρος μνήμης, ο οποίος σε διαφορετικές στιγμές μπορεί να χρησιμοποιηθεί από μεταβλητές διαφορετικού τύπου και μεγέθους. Η ένωση ορίζεται κατά τρόπο παρόμοιο με τη δομή:

```
union   <όνομα ένωσης>
{
     <τύπος δεδομένων>   <όνομα 1ου μέλους>;
     <τύπος δεδομένων>   <όνομα_2ου μέλους>;
     . . . . . .
     <τύπος δεδομένων>   <όνομα τελευταίου μέλους>;
};
```

Για να δηλωθεί μία μεταβλητή τύπου ένωσης, είτε τοποθετούμε το όνομα στο τέλος του ορισμού:

```
union unionExample
{
   char charVar;
   float floatVar;
   double doubleVar;
} unionVar;
```

είτε χρησιμοποιούμε ξεχωριστή δήλωση:

```
union unionExample unionVar;
```

Και τα τρία μέλη της ένωσης μοιράζονται τον ίδιο χώρο. Με τη δήλωση ο μεταγλωττιστής δεσμεύει τον χώρο, που απαιτεί ο μεγαλύτερος εκ των τύπων των μελών. Προφανώς, δεν επιτρέπεται η ταυτόχρονη

αποθήκευση δύο ή περισσοτέρων μελών. Η μεταβλητή **unionVar** καταλαμβάνει **8** bytes, καθόσον ο μεγαλύτερος τύπος εκ των τριών μελών είναι **double**.

Η αναφορά σε μέλος της ένωσης γίνεται με τον τελεστή τελεία (**.**). Με την πρόταση

```
unionVar.charVar='f';
```

ανατίθεται στο μέλος **charVar** η τιμή **'f'**, η οποία θα αποθηκευτεί στο πρώτο από τα **8** διαθέσιμα bytes.

Εάν ακολουθήσει η πρόταση

```
unionVar.floatVar=-12.346;
```

θα ανατεθεί στο μέλος **floatVar** η τιμή **=-12.346**, η οποία θα αποθηκευτεί στα πρώτα **4** bytes, σβήνοντας προφανώς το περιεχόμενο της **charVar**.

Θα πρέπει να σημειωθεί ότι, προφανώς, μόνο το πρώτο μέλος μπορεί να αρχικοποιηθεί μαζί με τη δήλωση μίας μεταβλητής τύπου ένωσης, καθόσον όλα τα μέλη μίας ένωσης αποθηκεύονται στον ίδιο χώρο μνήμης,:

```
union unionExample unionVar={'f'};
```

Μπορεί να δηλωθεί δείκτης σε ένωση:

```
union unionExample *unionPtr;
unionPtr=&unionExample;
```

οπότε τα μέλη θα προσπελαύνονται με χρήση του τελεστή βέλος:

```
unionPtr->charVar='F';     ή     unionPtr->doubleVar=345.32;
```

Ο κύριος λόγος χρήσης των ενώσεων είναι η εξοικονόμηση μνήμης.

8.10 Παράδειγμα ανάπτυξης προγράμματος

Στην παρούσα ενότητα θα αναπτυχθεί πρόγραμμα, με πυρήνα τύπους δεδομένων οριζόμενους από τον χρήστη, με βάσει τις αρχές του διαδικαστικού προγραμματισμού. Το πρόγραμμα θα λαμβάνει από το πληκτρολόγιο τον αριθμό των εργαζομένων σε μία επιχείρηση και θα δημιουργεί πίνακα για την καταχώριση των στοιχείων του κάθε εργαζομένου. Τα στοιχεία θα δίνονται από το πληκτρολόγιο και θα περιλαμβάνουν τις ακόλουθες πληροφορίες:

- **Ονοματεπώνυμο**: Όνομα και επώνυμο εργαζομένου (ξεχωριστά).
- **Διεύθυνση**: Όνομα οδού, αριθμός οδού, πόλη, ταχυδρομικός κώδικας.
- **Στοιχεία επικοινωνίας**: Τηλέφωνο εργασίας, κινητό τηλέφωνο, email εργασίας.
- **Θέση**: Τίτλος, κωδικός αριθμός εργαζομένου, τομέας της επιχείρησης στον οποίο εργάζεται, αριθμός γραφείου, ονοματεπώνυμο προϊσταμένου, ημερομηνία πρόσληψης (ημέρα, μήνας, έτος), μισθός.

Όταν δίνεται κάποιος κωδικός αριθμός εργαζομένου από το πληκτρολόγιο, θα αναζητείται στον πίνακα. Αν υπάρχει, τότε θα εμφανίζονται στην οθόνη οι πληροφορίες του αντίστοιχου εργαζομένου, αλλιώς θα εμφανίζεται ένα σχετικό μήνυμα.

Με βάση τις ανωτέρω προδιαγραφές, το πρόβλημα μπορεί να μεριστεί σε υποπροβλήματα ως εξής:

1. Μείζον ζήτημα στην υλοποίηση του κώδικα είναι η δημιουργία του κατάλληλου τύπου δεδομένων, που θα σχετίζεται με τον εργαζόμενο. Ακολουθώντας μία ιεραρχική ανάπτυξη του τύπου δεδομένων για τον εργαζόμενο, οι τέσσερις κατηγορίες στοιχείων (**ονοματεπώνυμο**, **διεύθυνση**, **στοιχεία επικοινωνίας**, **θέση**) μπορούν να υλοποιηθούν σε δομές και να αποτελέσουν τα μέλη μίας δομής **employeeT**, δηλαδή:

```
typedef struct employeeT
{
  struct nmT nm;            /* Δομή nmT για το ονοματεπώνυμο */
  struct addressT addr;     /* Δομή addressT για τη διεύθυνση */
  struct teleEmailT teleMail;/*Δομή teleEmailT για τηλέφωνα/email */
```

```
  struct jobDescrT job;   /* Δομή jobDescrT για τη θέση εργασίας */
} emplT;
```

Η δομή **nmT** θα έχει ως μέλη το όνομα και το επώνυμο ξεχωριστά:

```
struct nmT
{
  char name[40],surname[40];
};
```

Η δομή **addressT** θα έχει τη μορφή που αναπτύχθηκε σε προηγούμενα παραδείγματα:

```
struct addressT
{
  char streetName[40],city[40];
  int streetNo,zipCode;
};
```

Για τα τηλέφωνα της δομής **teleEmailT** ισχύει ο σχολιασμός του παραδείγματος 8.6.2 περί χρήσης αλφαριθμητικών:

```
struct teleEmailT
{
  char officeNo[15],homeNo[15],officeMail[40];
};
```

Η δομή **jobDescrT**, η οποία θα περιγράφει τη θέση του εργαζομένου στην επιχείρηση, είναι πιο σύνθετη από τις προηγούμενες τρεις, καθώς η ημερομηνία πρόσληψης – όπως καθορίστηκε στις προδιαγραφές – θα αποτελείται από τρία μέλη (ημέρα, μήνας έτος), επομένως θα υλοποιηθεί με μία δομή.

```
struct hiredateT
{
  int year,month,day;
};
```

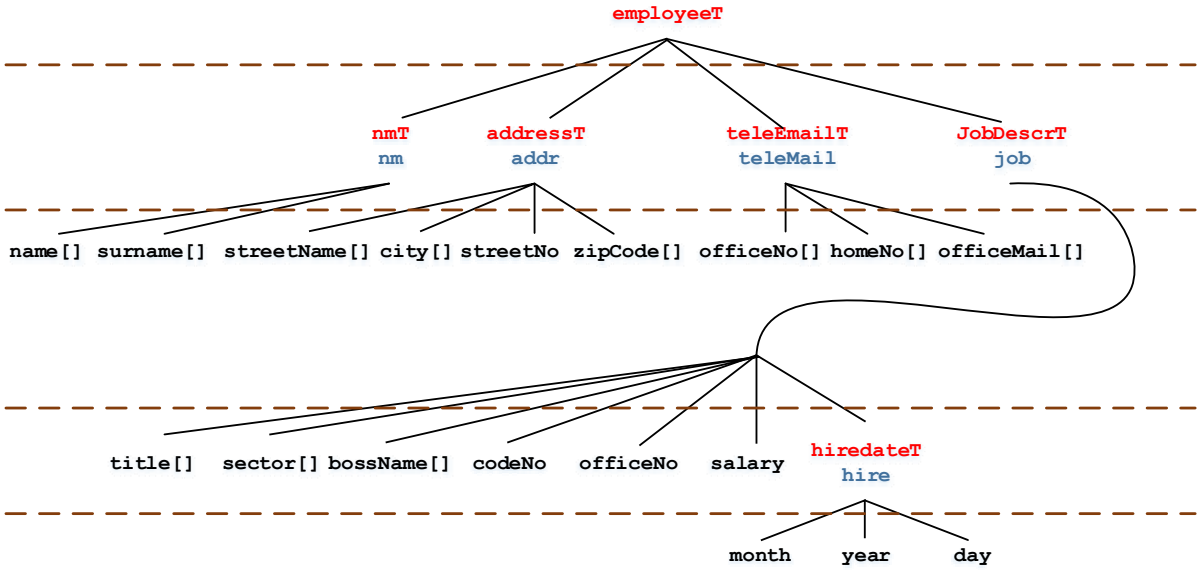
Οπότε η **jobDescrT** λαμβάνει την ακόλουθη μορφή:

```
struct jobDescrT
{
  char title[40],sector[40],bossName[40];
  int codeNo,officeNo,salary;
  struct hiredateT hire;
};
```

Με βάση τα παραπάνω, η δομή **employeeT** απεικονίζεται σε δεντρική μορφή στο **Σχήμα 8.1**. Με μαύρο χρώμα σημειώνονται οι μεταβλητές– μέλη που θα λαμβάνουν τιμές, με κόκκινο χρώμα οι τύποι δεδομένων και με γαλάζιο οι μεταβλητές– μέλη που είναι κι αυτές δομές. Οι καφέ διακεκομμένες γραμμές απεικονίζουν τα επίπεδα ένθεσης της δομής.

2. Εφόσον ο αριθμός των εργαζομένων, **size**, παρέχεται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματος, ο πίνακας με τα στοιχεία των εργαζομένων θα είναι ένας μονοδιάστατος δυναμικός πίνακας ***pArr** με στοιχεία τύπου **employeeT**, που θα δημιουργείται με δυναμική δέσμευση μνήμης:

```
pArr=(employeeT *)malloc(size*sizeof(employeeT));
assert(pArr!=NULL);
```



Σχήμα 8.1 *Δενδρικό διάγραμμα της δομής employeeT*

3. Σε ό,τι αφορά την ανάγνωση των δεδομένων, δημιουργείται μία συνάρτηση **`void getEmployee(emplT *ptr)`**, η οποία θα καλείται για κάθε εργαζόμενο από τη **`main()`** με την πρόταση

```
getEmployee(&pArr[i]);
```

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

```
void getEmployee(emplT *ptr)
{
  printf( "\n\t\tAdd new employee:\n\n" );

  printf( "\nEmployee's name:  " );   scanf( "%s",ptr->nm.name );
  printf( "\nSurname:  " );   scanf( "%s",ptr->nm.surname );

  printf( "\n\t\nEmployee's address:" );
  printf( "\nStreet name:  " );
  scanf( "%s",ptr->addr.streetName );
  ......  /* λοιπά στοιχεία διεύθυνσης */

  printf( "\n\t\nEmployee's phone numbers and email:" );
  printf( "\nOffice phone number:  " );
  scanf( "%s",ptr->teleMail.officeNo );
  ......  /* λοιπά στοιχεία τηλεφώνων και email */

  printf( "\n\t\nEmployee's job details:" );
  printf( "\nCode number:  " );     scanf( "%d",&ptr->job.codeNo );
  printf("\nSalary:  ");    scanf("%d",&ptr->job.salary);
  printf("\nYear of recruitment:  ");
  scanf("%d",&ptr->job.hire.year);
  ......  /* λοιπά στοιχεία της θέσης εργασίας */
}
```

4. Για την αναζήτηση ενός εργαζομένου βάσει του κωδικού του δημιουργείται η συνάρτηση

```
void searchEmployee(int code, emplT *ptr, int size);
```

η οποία δέχεται ως ορίσματα τον κωδικό αναζήτησης, το περιεχόμενο του δείκτη **pArr** και το μέγεθος του πίνακα. Η συνάρτηση σαρώνει τον πίνακα των εργαζομένων μέσω επαναληπτικής πρότασης και στην πρώτη εύρεση του κωδικού καλεί τη συνάρτηση **void printEmployee(employeeT *ptr)**, για να εκτυπωθούν τα στοιχεία του ανευρεθέντος εργαζομένου. Σε περίπτωση αρνητικού αποτελέσματος στην αναζήτηση, εμφανίζεται σχετικό μήνυμα. Ο κώδικας της συνάρτησης **searchEmployee()** είναι ο ακόλουθος:

```
void searchEmployee(int code, emplT *ptr, int size)
{
  int j=0,index=-1;
  while ((j<size) && (index==-1))
  {
    if (code==ptr[j].job.codeNo) index=j;
    j++;
  }
  printf("\n\t\tSearch results:\n");
  if (index==-1)
    printf( "\nThe given employee code does not match to an exist-
ing one\n" );
  else
    printEmployee(&ptr[index]);
}
```

5. Ο κώδικας της συνάρτησης **printEmployee(emplT *ptr)** είναι παρόμοιος με εκείνον της **getEmployee()**, με μόνη διαφορά την αντικατάσταση των **scanf()** με **printf()**, μέσω των οποίων τυπώνονται τα δεδομένα.

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

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

struct hiredateT
{
  int year,month,day;
};
/*-------------------------------*/
struct jobDescrT
{
  char title[40],sector[40],bossName[40];
  int codeNo,officeNo,salary;
  struct hiredateT hire;
};

struct nmT
{
  char name[40],surname[40];
};

struct addressT
{
  char streetName[40],city[40];
  int streetNo,zipCode;
};
```

```
struct teleEmailT
{
  char officeNo[15],homeNo[15],officeMail[40];
};
/*--------------------------------*/
typedef struct employeeT
{
   struct nmT nm;
   struct addressT addr;
   struct teleEmailT teleMail;
   struct jobDescrT job;
} emplT;
/*--------------------------------*/
void printEmployee(emplT *ptr);
void getEmployee(struct employeeT *ptr);
void searchEmployee(int code, struct employeeT *ptr, int size);
/*--------------------------------*/
int main()
{
   int i,size,code;
   struct employeeT *pArr;

   printf( "Give the number of employees: " );
   scanf( "%d",&size );
   pArr=(struct employeeT *)malloc(size*sizeof(emplT));
   assert(pArr!=NULL);

   for (i=0;i<size;i++)
      getEmployee(&pArr[i]);

   printf( "\n\n     Give an employee's code number: " );
   scanf( "%d",&code );
   searchEmployee(code,pArr,size);

   free(pArr);
   return 0;
 }
/*--------------------------------*/
void getEmployee(emplT *ptr)
{
   printf( "\n\t\tAdd new employee:\n\n" );

   printf( "\nEmployee's name:  " );
   scanf( "%s",ptr->nm.name );
   printf( "\nSurname:  " );
   scanf( "%s",ptr->nm.surname );

   printf( "\t\nEmployee's address:" );
   printf( "\nStreet name:  " );
   scanf( "%s",ptr->addr.streetName );
   ......  /* λοιπά στοιχεία διεύθυνσης */

   printf( "\t\nEmployee's phone numbers and email:" );
   printf( "\nOffice phone number:  " );
   scanf( "%s",ptr->teleMail.officeNo );
```

```
        ......  /* λοιπά στοιχεία τηλεφώνων και email */

        printf( "\t\nEmployee's job details:" );
        printf( "\nCode number:  " );
        scanf( "%d",&ptr->job.codeNo );
        printf("\nSalary:  ");
        scanf("%d",&ptr->job.salary);
        printf("\nYear of recruitment:  ");
        scanf("%d",&ptr->job.hire.year);
        ......  /* λοιπά στοιχεία της θέσης εργασίας */
      }
      /*--------------------------------*/
      void printEmployee(emplT *ptr)
      {
        printf( "\n\t\tDetails of the employee with code number %d:\n\n",
ptr->job.codeNo );
        printf( "\nEmployee's name:  %s",ptr->nm.name );
        printf( "\nSurname:  %s",ptr->nm.surname );
        printf( "\n\t\nEmployee's address:" );
        printf( "\nStreet name:  %s",ptr->addr.streetName );
        ......  /* λοιπά στοιχεία διεύθυνσης */
        printf( "\n\t\nEmployee's phone numbers and email:" );
        printf( "\nOffice phone number:  %s",ptr->teleMail.officeNo );
        ......  /* λοιπά στοιχεία τηλεφώνων και email */
        printf( "\n\t\nEmployee's job details:" );
        printf("\nSalary:  %d",ptr->job.salary);
        printf("\nYear of recruitment:  %d",ptr->job.hire.year);
        ......  /* λοιπά στοιχεία της θέσης εργασίας */
      }
      /*--------------------------------*/
      void searchEmployee(int code, struct employeeT *ptr, int size)
      {
        int j=0,index=-1;
        while ((j<size) && (index==-1))
        {
          if (code==ptr[j].job.codeNo)
            index=j;
          j++;
        }
        printf("\n\n\t\tSearch results:\n");
        if (index==-1)
          printf( "\nThe given employee code does not match to an exist-
ing one\n" );
        else
          printEmployee(&ptr[index]);
      }
```

```
Give the number of employees: 2

                    Add new employee:


Employee's name:       George

Surname:       Pappas

Employee's address:
Street name:    Hnioxou

Employee's phone numbers and email:
Office phone number:  +302102222222

Employee's job details:
Code number:  123

Salary:  1234

Year of recruitment:    2009

                    Add new employee:


Employee's name:       John

Surname:       Papadopoulos

Employee's address:
Street name:    Magnesias

Employee's phone numbers and email:
Office phone number:  +302321022222

Employee's job details:
Code number:  345

Salary:  1567

Year of recruitment:    2002

        Give an employee's code number: 123

                    Search results:

                    Details of the employee with code number 123:


Employee's name:       George
Surname:       Pappas

Employee's address:
Street name:    Hnioxou

Employee's phone numbers and email:
Office phone number:  +302102222222

Employee's job details:
Salary:  1234
Year of recruitment:    2009
```

Εικόνα 8.6 *Η έξοδος του προγράμματος*

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Ποια από τις ακόλουθες προτάσεις είναι λαναθασμένη;

(α) `enum week_days {Sun=30, Mon, Tue, Wed, Thu, Fri, Sat};`

(β) `enum week_days {Sun=30, Mon, Tue, Wed=45, Thu, Fri, Sat};`

(γ) `enum week_days {Sun=30, Mon, Tue, Wed=-1, Thu, Fri, Sat};`

(δ) `enum week_days={Sun=30, Mon, Tue, Wed, Thu, Fri, Sat};`

(2) Ποιο είναι το αποτέλεσμα του ακόλουθου προγράμματος;

```
#include <stdio.h>
enum week_days {Sun=30, Mon, Tue, Wed=-1, Thu, Fri, Sat};
int main()
{
   week_days day;
   printf("Sun=%d, Sat=%d\n",Sun,Sat);
   return 0;
}
```

(α) Θα εμφανιστούν σφάλματα κατά τη μεταγλώττιση/αποσφαλμάτωση.

(β) Sun=30, Sat=2

(γ) Sun, Sat

(δ) Sun=30, Sat=36

(3) Τα μέλη μίας δομής μπορούν να είναι:

(α) οι βασικοί τύποι δεδομένων (`int`, `char`, `float`, `double`).

(β) απαριθμητικοί τύποι, οι οποίοι θα πρέπει να οριστούν αμέσως μετά τον ορισμό της δομής.

(γ) πίνακες.

(δ) άλλες δομές, οι οποίες θα πρέπει να έχουν οριστεί πριν τον ορισμό της δομής.

(4) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;

(α) Η λέξη κλειδί `struct` προσδιορίζει την αρχή του ορισμού μίας δομής και δημιουργεί έναν νέο τύπο.

(β) Ο ορισμός δομής τελειώνει προαιρετικά με ερωτηματικό (`;`).

(γ) Ο ορισμός μίας δομής δεν δεσμεύει χώρο στη μνήμη για μεταβλητές, αλλά απλώς δημιουργεί έναν νέο τύπο.

(δ) Ο ορισμός ενός τύπου δομής προηγείται της δήλωσης μεταβλητών αυτού του τύπου.

(5) Ποια από τις ακόλουθες λειτουργίες δεν μπορεί να εκτελεστεί σε μία δομή;

(α) Ανάθεση μεταβλητών του ίδιου τύπου, δηλαδή εάν `address1` και `address2` είναι δύο μεταβλητές τύπου δομής `addrT`, με την ανάθεση `address1=address2` τα μέλη της `address1` αποκτούν τις τιμές των αντίστοιχων μελών της `address2`.

(β) Αντιγραφή μεταβλητών δομής με χρήση της συνάρτησης `strcpy()`, δηλαδή εάν `address1` και `address2` είναι δύο μεταβλητές τύπου δομής `addrT`, με την εντολή `strcpy(address1,address2)` τα μέλη της `address1` αποκτούν αντίγραφα των τιμών των αντίστοιχων μελών της `address2`.

(γ) Η διεύθυνση μίας μεταβλητής τύπου δομής μπορεί να ανατεθεί σε έναν δείκτη (pointer).

(δ) Χρήση του τελεστή `sizeof`, δηλαδή η εντολή `sizeof(struct address1)`, επιστρέφει τον αριθμό των bytes που απαιτούνται για την αποθήκευση στη μνήμη μίας μεταβλητής τύπου δομής `addrT`.

Ασκήσεις

Άσκηση 1

Να περιγραφεί αναλυτικά η λειτουργία του ακόλουθου προγράμματος, να δοθούν τα αποτελέσματά του και να απεικονιστεί ο χάρτης μνήμης.

```
#include <stdio.h>

struct card
{
   char *x, *y;
};

int main()
{
   card aCard,*pcard;
   aCard.x="Athens";
   aCard.y="Thessaloniki";
   pcard = &aCard;
   printf( "%s%s%s\n", pcard->x," to ",pcard->y );
   printf("%s%s%s\n", (*pcard).x," to ",(*pcard).y);
}
```

Άσκηση 2

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

(i) **Θα δημιουργεί τη δομή `metricT` με μέλη τους ακεραίους `feet` και `inches`.**

(ii) **Θα καλείται η συνάρτηση `void readMetric(metricT *pengl)`, η οποία θα διαβάζει το ύψος σε πόδια και ίντσες και θα ενημερώνει έμμεσα μία μεταβλητή της `main()` τύπου `metricT` και με όνομα `height`.**

(iii) **Ακολούθως, το ύψος θα μετατρέπεται σε εκατοστά με ακρίβεια δεύτερου δεκαδικού ψηφίου και θα αποτυπώνεται στην οθόνη.**

Δίνεται ότι ένα πόδι έχει 12 ίντσες και μία ίντσα αντιστοιχεί σε 2.54 εκατοστά.

Άσκηση 3

Να περιγραφεί αναλυτικά η λειτουργία του ακόλουθου προγράμματος, να απεικονιστούν οι θέσεις μνήμης που αυτό καταλαμβάνει και να δοθούν τα αποτελέσματα της εκτέλεσής του:

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

struct domi
{
   char x[40],y[40];
};

domi func(char *mptr);

int main()
{
   domi d1,d2={"One","Two"};
   char in[40];
   strcpy(in,d2.y);
   d1=func(in);
   printf( "\n\td1.x=%s\n\n",d1.y );
   d1=func(d2.x);
   printf( "\n\td1.y=%s\n\n",d1.y );
```

```
}
domi func(char *mptr)
{
   domi c1={"Three","Four"};
   static char w[40]="Start";
   strcat(w,mptr);
   strcpy(c1.y,w);
   return (c1);
}
```

Άσκηση 4

(α) Να δημιουργηθεί η δομή `paral`, που θα αποτελείται από τις πραγματικές μεταβλητές `a`, `b`.

(β) Να γραφούν η συνάρτηση `float paralArea(paral *c)`, η οποία θα επιστρέφει στην κύρια συνάρτηση το εμβαδόν του παραλληλογράμμου, το οποίο περιγράφεται από τη μεταβλητή, στην οποία δείχνει η τυπική παράμετρος της συνάρτησης.

γ) Να γραφεί ο κώδικας της `main()`, μέσα στην οποία θα δημιουργείται η μεταβλητή τύπου `paral` με όνομα `pr`, στην οποία θα αποδίδονται τιμές από το πληκτρολόγιο. Ακολούθως, θα καλείται η συνάρτηση με πραγματική παράμετρο τη διεύθυνση της `pr` και θα αποδίδεται στη μεταβλητή `area` το εμβαδό, που θα προκύψει από την συνάρτηση `paralArea()`.

Άσκηση 5

Να δημιουργηθεί ο τύπος δεδομένου δομής `timeT`, που θα αποτελείται από τις τρεις ακέραιες μεταβλητές `hour`, `min`, `sec`. Στη συνάρτηση `main()` θα δημιουργηθεί μεταβλητή `tme` τύπου `timeT`, η οποία θα λάβει τιμές από το πληκτρολόγιο. Ακολούθως, θα κληθεί η συνάρτηση `void TimeInSec( timeT *t)`, η οποία θα μετατρέπει σε δευτερόλεπτα τον χρόνο που ορίζει η μεταβλητή του ορίσματός της, τον οποίο και θα εκτυπώνει.

Άσκηση 6

Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:

(α) Θα ορίζει μία δομή `vectT` με μέλη πραγματικούς αριθμούς `x` και `y`.

(β) Θα δεσμεύει μνήμη για μονοδιάστατο πίνακα `arr`, μεγέθους `n` και με στοιχεία τύπου `vectT`. Το μέγεθος θα καθορίζεται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματος. Η δεσμευθείσα μνήμη θα αποδεσμεύεται στο τέλος της εκτέλεσης του προγράμματος.

(γ) Θα καλείται η συνάρτηση `void readData(vectT *parr, int n)`, η οποία θα αποδίδει τιμές στον `arr` (έμμεσα) από το πληκτρολόγιο.

(δ) Θα καλείται η συνάρτηση `void stats(vectT *parr, int n)`, η οποία για το μέλος `x` της δομής `vectT` θα υπολογίζει τη μέγιστη, την ελάχιστη και τη μέση τιμή των στοιχείων του `arr`. Ο πίνακας `arr` και οι στατιστικές τιμές θα εμφανίζονται στην οθόνη.

Άσκηση 7

Να δημιουργηθεί ο τύπος δεδομένου δομής `circleT`, που έχει ως μέλη τις τρεις πραγματικές μεταβλητές `x` (τετμημένη του κέντρου του κύκλου), `y` (τεταγμένη του κέντρου του κύκλου), `r` (ακτίνα του κύκλου). Να γραφούν οι ακόλουθες συναρτήσεις :

- **`void readCircle(circleT *pc)`, η οποία θα δέχεται από το πληκτρολόγιο τιμές για τα μέλη της μεταβλητής, στην οποία δείχνει ο δείκτης `pc`.**
- **`void printCircle(circleT *pc)`, η οποία θα εμφανίζει στην οθόνη τα περιεχόμενο της μεταβλητής, στην οποία δείχνει ο δείκτης `pc`.**
- **`float circleArea(circleT *pc)`, η οποία θα επιστρέφει στην καλούσα συνάρτηση το εμβαδόν του κύκλου που περιγράφεται από τη μεταβλητή, στην οποία δείχνει ο δείκτης `pc`.**

Να γραφεί ο κώδικας της `main()`, μέσα στην οποία θα δημιουργούνται οι μεταβλητές τύπου `circleT` με όνομα `cir` και τύπου `float` με όνομα `area`, και θα καλούνται διαδοχικά οι τρεις

συναρτήσεις `readCircle(&cir)`, `printCircle(&cir)`, `area=circleArea(&cir)`. Ακολούθως, η τιμή του εμβαδού θα εμφανίζεται στην οθόνη.

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

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

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

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Horton, I. (2006), *Beginning C – from Novice to Professional*, 4th ed., Apress.

Prinz, P. & Crawford, T. (2005), *C in a Nutshell*, O'Reilly.

9. Αρχεία

Σύνοψη
Στο κεφάλαιο αυτό μελετώνται τα αρχεία. Αρχικά, ο αναγνώστης εισάγεται στις έννοιες των καναλιών ή ροών και της ενδιάμεσης μνήμης και δίνεται ο ορισμός του αρχείου στη γλώσσα C. Μελετώνται τα αρχεία κειμένου και τα δυαδικά αρχεία, όπου περιγράφονται οι λειτουργίες ανοίγματος/ κλεισίματος, ανάγνωσης και εγγραφής δεδομένων. Ακολούθως, παρουσιάζεται η τυχαία προσπέλαση δυαδικού αρχείου και η ανάγνωση/ εγγραφή ανά γραμμή. Στην τελευταία ενότητα παρουσιάζεται ένα εκτενές παράδειγμα ανάπτυξης προγράμματος, στο οποίο υλοποιείται πλήθος λειτουργιών των αρχείων και γίνεται χρήση εννοιών και εργαλείων που αναπτύχθηκαν στα προηγούμενα κεφάλαια.

Λέξεις κλειδιά
κανάλια (ροές) – αρχείο κειμένου – δυαδικό αρχείο – δείκτης αρχείου – fopen – fclose – fprintf – fscanf – fgets – fputs – getc – putc – fread – fwrite – EOF – feof – τυχαία προσπέλαση – fseek – ftell.

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C – μεταβλητές – εκφράσεις – τελεστές – έλεγχος ροής προγράμματος – συναρτήσεις – πίνακες – δείκτες – δυναμική διαχείριση μνήμης – δομές.

9.1 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 μνήμες flash, τα cd/dvd κ.λπ. 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 γλώσσα 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**.

9.1.1 Τα κανάλια stdin, stdout, stderr

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

1. 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`.

2. 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» `stdout`.

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.

Όταν εκτελείται ένα πρόγραμμα **program_name.c**, με την εντολή **program_name < filename** ορίζεται ως η καθιερωμένη είσοδος αντί του πληκτρολογίου το αρχείο **filename**. Αντίστοιχα, με την εντολή **program_name > filename** ορίζεται ως κύρια έξοδος αντί για την οθόνη το αρχείο **filename**. Οι εντολές αυτές δίνονται από τη γραμμή διαταγής (command line).

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

9.1.2 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 έναν χαρακτήρα από ένα αρχείο, στην πράξη διαβάζεται όλο το μπλοκ, στο οποίο βρίσκεται αποθηκευμένος ο χαρακτήρας. Έτσι, με τη χρήση του buffer εάν χρειαστούμε άλλους χαρακτήρες από το ίδιο μπλοκ, δεν επιστρέφουμε στη συσκευή, αλλά τους διαβάζουμε από τον 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 αρχείο κεφαλίδας **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 στον συμβολισμό, έχει επικρατήσει τα ονόματα των δεικτών αρχείου να αρχίζουν από ***f*** (**f**ile).

9.1.3 Κατηγορίες αρχείων

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

- Τα **δυαδικά αρχεία** (binary files), τα οποία αποθηκεύουν όλους τους τύπους δεδομένων: **char**, **int**, **float**, **double**, δομή, απαριθμητικό τύπο κ.λπ. Ο τρόπος αποθήκευσης είναι ίδιος με εκείνον της κύριας μνήμης, δηλαδή δε γίνεται μεταγλώττιση των bytes αλλά απλώς διαβάζονται και γράφονται 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 κειμενογράφους (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 (ένα για κάθε χαρακτήρα κι ένα για τον χαρακτήρα τερματισμού **'\0'**).

Τα αρχεία κειμένου είναι αναγνώσιμα από τους συντάκτες. Μάλιστα, τα προγράμματα της γλώσσας 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.

9.2 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:

```
fp=fopen(filename, mode);
```

όπου ο δείκτης αρχείου **fp** έχει δηλωθεί προηγουμένως με τη δήλωση **FILE *fp;**

- Η συνάρτηση **fopen()** δεσμεύει τους απαραίτητους πόρους από το λειτουργικό σύστημα, δημιουργεί το κανάλι επικοινωνίας και επιστρέφει στο πρόγραμμα που την κάλεσε έναν δείκτη **fp**, ο οποίος 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 **fp** χειρίζεται το αρχείο μέσα στο πρόγραμμα. Ένα από τα πεδία της δομής **FILE** είναι ο **δείκτης θέσης αρχείου** (file position indicator), ο οποίος δείχνει στο byte απ' όπου ο επόμενος χαρακτήρας πρόκειται να διαβαστεί ή όπου ο επόμενος χαρακτήρας πρόκειται να εγγραφεί.

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

```
"c:\\temporary\\c_folder\\myfile.txt"
```

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

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

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

```
fp=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 μηδενιστεί 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 μηδενιστεί και τα περιεχόμενα θα διαγραφούν.

- **a+**: Άνοιγμα υπάρχοντος αρχείου ή δημιουργία νέου σε μορφή προσάρτησης. 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(fp);
```

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, θεωρώντας τον δείκτη **fp** ως έναν δείκτη σε αρχείο:

```
fclose(fp);     σωστό
fclose(fp+1);   λάθος
```

2. Η συνάρτηση **fopen()** δεσμεύει μνήμη. Εάν αμεληθεί να απελευθερωθεί με χρήση της **fclose()**, θα υπάρξει διαρροή μνήμης. Για τον 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.

9.3 Ανάγνωση – εγγραφή χαρακτήρων σε αρχεία

9.3.1 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 *fp);
```

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

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

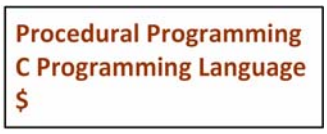
9.3.1.1 Παράδειγμα

Να καταστρωθεί πρόγραμμα, το οποίο διαβάζει χαρακτήρες από το πληκτρολόγιο και τους γράφει σε αρχείο, έως ότου πληκτρολογηθεί το σύμβολο του δολαρίου (**$**).

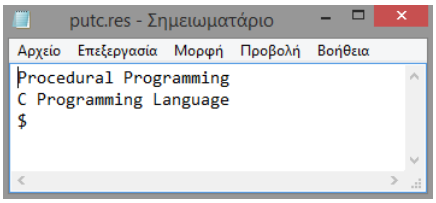
```
#include<stdio.h>

int main()
{
  FILE *fp;
  char ch;
  fp=fopen("putc.res","w");
  if (fp==NULL)
```

```
      printf( "\t\tFILE ERROR: Exit program\n" );
   else
   {
      do
      {
         ch=getchar();
         putc(ch,fp);
      } while (ch!='$');
   }
   fclose(fp);

   return 0;
}
```

Ακολουθούν η έξοδος στην οθόνη (οι χαρακτήρες που πληκτρολογήθηκαν) και το προκύπτον αρχείο **putc.res**. Για την προβολή των αρχείων θα χρησιμοποιείται ο κειμενογράφος *Σημειωματάριο* (*Notepad*).



Εικόνα 9.1.α *Η έξοδος του προγράμματος του παραδείγματος 9.3.1.1*



Εικόνα 9.1.β *Το προκύπτον αρχείο του προγράμματος του παραδείγματος 9.3.1.1*

9.3.2 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 *fp);
```

όπου `fp` είναι 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 αρχείο κεφαλίδας `stdio.h`.

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

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

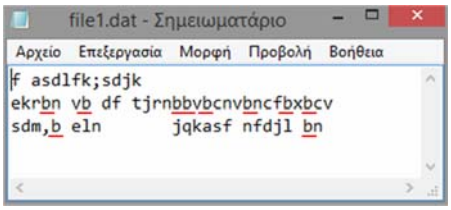
9.3.2.1 Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα βρίσκει πόσες φορές υπάρχει ο χαρακτήρας `'\0'` στο αρχείο **file1.dat**.

```
#include <stdio.h>

int  main()
{
```

```
  FILE *fp;
  char charVar;
  int sumb=0;
  fp=fopen("file1.dat","r");
  charVar=getc(fp);
  while (charVar!=EOF)
  {
    if (charVar=='b') sumb++;
    charVar=getc(fp);
  }
  fclose(fp);
  printf( "\nNumber of 'b' appearances: %d",sumb);

  printf("\n\n");

  return 0;
}
```

Η έξοδος του προγράμματος είναι: **Number of 'b' appearances: 10**, όπως προκύπτει από το περιεχόμενο του τυχαία διαμορφωθέντος αρχείου δεδομένων **file1.dat**.



Εικόνα 9.2 *Το αρχείο ανάγνωσης του προγράμματος του παραδείγματος 9.3.2.1*

9.3.2.2 Παράδειγμα

Το παρακάτω πρόγραμμα υπολογίζει τον αριθμό των λέξεων που περιέχονται σε ένα αρχείο ASCII. Το πρόγραμμα 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. Όλη αυτή τη συμβολοσειρά τη μετρά ως λέξη. Κατόπιν διαβάζει τους πραγματικούς χαρακτήρες έως την εμφάνιση του επόμενου λευκού χαρακτήρα.

Μία μεταβλητή (σημαία) ελέγχει κατά πόσον το πρόγραμμα βρίσκεται στο μέσο μίας λέξης ή στο μέσο κάποιου κενού. Το αρχείο ανάγνωσης δεδομένων είναι εκείνο του παραδείγματος 9.3.2.1.

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

int main()
{
  FILE *fptr;
  char ch,string[81];
  int white=1;    /* Σημαία λευκού χαρακτήρα */
  int count=0;    /* Μετρητής λέξεων */
  fptr=fopen("file1.dat","r");
  if (fptr==NULL)
  {
    printf( "ERROR: can't open file" );
    exit(1);
```

```
    }
    while ((ch=getc(fptr))!=EOF) /* Ανάγνωση χαρακτήρων έως EOF */
    {
       switch(ch)
       {
          /* Έλεγχος για λευκούς χαρακτήρες: τριπλή case με κοινό
             σώμα */
          case ' ':
          case '\t':
          case '\n':
             white++;
          break;
          default:  /* Μη λευκοί χαρακτήρες, μέτρηση λέξεων */
             /* Συνθήκη για αύξηση του μετρητή λέξεων και μηδενισμό της
                σημαίας, ώστε να ξεκινήσει εκ νέου η διαδικασία ελέγχου
                του τέλους μίας λέξης  */
             if (white)
             {
                white=0;
                count++;
             }
          break;
       }
    }
    fclose( fptr );
    printf( "The file contains %d words\n",count );

    return 0;
}
```

Η έξοδος του προγράμματος είναι: **The file contains 11 words**. Από την **Εικόνα 9.2** προκύπτει ότι όντως το αρχείο **file1.dat** περιέχει **11** λέξεις.

9.4 Μορφοποιούμενες συναρτήσεις εισόδου – εξόδου σε αρχεία

9.4.1 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( fp, ορίσματα );
```

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

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

9.4.1.1 Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

```c
#include <stdio.h>

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

   return 0;
}
```

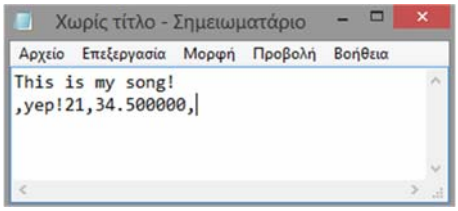
Αρχικά, ορίζεται η ακέραια μεταβλητή **cnt** και ο δείκτης αρχείου **fp**. Ακολουθεί ο ορισμός ενός δείκτη χαρακτήρων **filename**, ο οποίος δείχνει στο αλφαριθμητικό **testfile.txt**, και το αλφαριθμητικό **msg** που έχει αρχικοποιηθεί.

Ο δείκτης **fp** 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, καθώς τυπώνεται ως **34.500000**),
- το κόμμα και την αλλαγή γραμμής (**2** bytes).

Η έξοδος του προγράμματος είναι: **Number of bytes written in testfile.txt: 36**.

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



Εικόνα 9.3 *Το αρχείο εγγραφής του προγράμματος του παραδείγματος 9.3.2.2*

9.4.2 Η συνάρτηση fscanf

Η συνάρτηση **fscanf()** χρησιμοποιείται για ανάγνωση δεδομένων 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:

```c
fscanf( pF, ορίσματα );
```

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

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

9.4.2.1 Παράδειγμα

Να γραφεί πρόγραμμα, το οποίο θα δημιουργεί μονοδιάστατο πίνακα τριών θέσεων, 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 προϋπάρχον 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

struct structTypeT;
{
  char nm[40];
  char srnm[40];
  char phNo[15];
};

int main()
{
  struct structTypeT id[N];
  FILE *f1,*f2;
  int i;

  f1=fopen("file1.dat","r");     assert( f1!=NULL );
  f2=fopen( "file2.dat","w" );
  for (i=0; i<N; i++)
  {
    fscanf( f1,"%s %s %s\n",id[i].nm,id[i].srnm,id[i].phNo );
    fprintf( f1,"%s %s %s\n",id[i].nm,id[i].srnm,id[i].phNo );
  }
  fclose(f2);
  close(f1);

  return 0;
}
```

9.5 Ανάγνωση – εγγραφή σε δυαδικά αρχεία

Αν και η χρήση των **fprintf()**, **fscanf()** είναι συχνά ο πιο εύκολος τρόπος, για να γράφουμε σε αρχείο ή να διαβάζουμε από ένα αρχείο, 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()**.

9.5.1 Η συνάρτηση fread

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

```
int fread(void *buffer, int length, int numItems, 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 χρησιμοποιείται η **sizeof**.
- **numItems** 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.

Αξίζει 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.

9.5.1.1 Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

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

int main()
{
  FILE *fp;
  int buf[40],i,cnt,n=24;
  fp=fopen("file.dat","rb");  assert(fp!=NULL);
  cnt=fread(buf,sizeof(int),n,fp);
  if (cnt!=n)
  {
    printf( "ERROR");
    exit(-1);
  }
  printf( "Number of items read:  %d\n",cnt );
  fclose(fp);

  return 0;
}
```

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 και η **cnt** γίνεται ίση με τη

n. Σε περίπτωση σφάλματος έχει ληφθεί πρόνοια για έξοδο από το πρόγραμμα. Στο τέλος το προγράμματος κλείνει το αρχείο **file.dat**. Η έξοδος του προγράμματος είναι: **Number of items read: 24**.

9.5.2 Η συνάρτηση fwrite

Η συνάρτηση **fwrite()** χρησιμοποιείται για την εκτύπωση μπλοκ δεδομένων σε ένα αρχείο. Ορίζεται στο αρχείο κεφαλίδας **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 στον **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 (τον 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 **fwrite**/**fwrite()** μπορούν να χρησιμοποιηθούν με τον ίδιο τρόπο και σε αρχεία κειμένου, όπου κάθε χαρακτήρας θα διαβάζεται/ εγγράφεται ως ξεχωριστό δεδομένο.

9.5.2.1 Παράδειγμα

Να περιγραφεί η λειτουργία του ακόλουθου προγράμματος:

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

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

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

  return 0;
}
```

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 τη **n**, όπως φαίνεται στην έξοδο στην οθόνη. Σε περίπτωση σφάλματος γίνεται άμεση έξοδος από το πρόγραμμα. Στο τέλος το προγράμματος κλείνει το αρχείο **file.dat**. Η έξοδος του προγράμματος είναι: **Number of items written: 32**.

9.5.2.2 Παράδειγμα

Να γραφεί κώδικας για το παράδειγμα 9.4.2.1 με χρήση δυαδικών αρχείων και των συναρτήσεων **fread()**/**fwrite()**.

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

#define N 3

struct structTypeT;
{
  char nm[40];
  char srnm[40];
  char phNo[15];
};

int main()
{
  structTypeT 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);

  return 0;
}
```

9.5.2.3 Παράδειγμα

Στο πρόγραμμα που ακολουθεί διαβάζονται χαρακτήρες από το πληκτρολόγιο και εγγράφονται σε δυαδικό αρχείο, το οποίο λόγω της φύσης των περιεχομένων του (χαρακτήρες) είναι αναγνώσιμο με τους συντάκτες κειμένου. Ως συνθήκη τερματισμού θεωρείται η ανάγνωση του δολαρίου (**$**).

Ακολούθως, με χρήση της συνάρτησης

```
void readCharacter(FILE *fp, char *ps, int k);
```

διαβάζεται ο **14ος** χαρακτήρας, χρησιμοποιώντας τυχαία προσπέλαση του αρχείου.

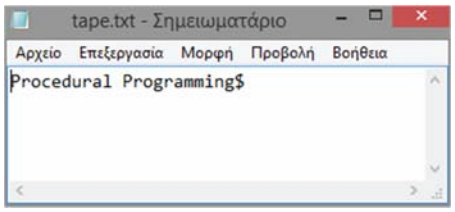
Η συνάρτηση

```
void saveCharacter(FILE *fp, char *ps, int k)
```

τοποθετεί στη **14η** θέση του αρχείου το θαυμαστικό (**!**).

```
#include <stdlib.h>
#include <stdio.h>
#define NAME "tape.txt"
void readCharacter(FILE *fp, char *ps, int k );
void saveCharacter(FILE *fp, char *ps, int k );
int main()   {
  FILE *fp;
  char charVar;
  fp=fopen(NAME,"wb");
  do
  {
     charVar=getchar();
     fwrite(&charVar,sizeof(char),1,fp);
  } while (charVar!='$');
  fclose(fp);
  fp=fopen(NAME,"rb+");
  readCharacter(fp,&charVar,13);
  printf("\n\nThe 14th character is: %c\n",charVar);
  charVar='!';
  saveCharacter(fp,&charVar,13);
  fclose(fp);
  return 0;
}
/*-----------------------------------*/
void readCharacter(FILE *fp, char *ps, int k )
{
  fseek(fp,k*sizeof(char),SEEK_SET);
  fread(ps,sizeof(char),1,fp);
}
/*-----------------------------------*/
void saveCharacter(FILE *fp, char *ps, int k )
{
  fseek(fp,k*sizeof(char),SEEK_SET);
  fwrite(ps,sizeof(char),1,fp);
}
```

Μετά την εγγραφή των δεδομένων και το πρώτο κλείσιμο του αρχείου **tape.txt**, τα περιεχόμενα του αρχείου απεικονίζονται στην **Εικόνα 9.4**, απ' όπου προκύπτει ότι ο **14ος** χαρακτήρας είναι το **'o'**.



Εικόνα 9.4 *Το αρχείο εγγραφής του προγράμματος του παραδείγματος 9.5.2.3 πριν τη μεταβολή του 14ου χαρακτήρα*

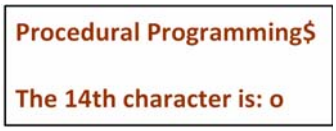
Ακολούθως, το αρχείο **tape.txt** ανοίγει εκ νέου σε κατάσταση **rb+**, επομένως μπορούν και να γραφούν και να αναγνωστούν δεδομένα. Η κλήση της συνάρτησης

```
readCharacter(fp,&charVar,13);
```

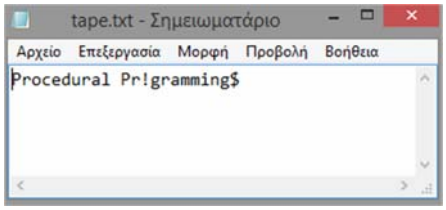
οδηγεί στην ανέρευση του **14ου** χαρακτήρα και η κλήση της συνάρτησης

```
saveCharacter(fp,&charVar,13);
```

οδηγεί στην αντικατάστση του **14ου** χαρακτήρα. Η έξοδος του προγράμματος απεικονίζεται στην **Εικόνα 9.5.α** και η τελική μορφή του αρχείου – με τον αντικατασταθέντα χαρακτήρα – παρουσιάζεται στην **Εικόνα 9.5.β**.



Εικόνα 9.5.α *Η έξοδος του προγράμματος του παραδείγματος 9.5.2.3*



Εικόνα 9.5.β *Το προκύπτον αρχείο του προγράμματος του παραδείγματος 9.5.2.3*

9.5.3 Η συνάρτηση feof

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

9.6 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 οποιονδήποτε αριθμό χαρακτήρων. Η λειτουργία θα περιγραφεί με τη βοήθεια του ακόλουθου τμήματος κώδικα:

```
FILE *fpin,*fpout;
char buf[100];
int cnt;
fpin=fopen("src.txt","r");
fpout=fopen("dest.txt","w");
if (fpin==NULL)
  exit(-1);
cnt=fread(buf,sizeof(char),100,fpin);
while(cnt==100)
{
```

```
    fwrite(buf,sizeof(char),100,fpout);
    cnt=fread(buf,sizeof(char),100,fpin);
  }
  if (cnt!=0)
    fwrite(buf,sizeof(char),cnt,fpout);
  fclose(fpout);
  fclose(fpin);
```

Αρχικά, ορίζονται οι δείκτες αρχείου **fpin** και **fpout** και ο **buffer** χαρακτήρων **100** θέσεων. Ο **fpin** χρησιμοποιείται, για να ανοίξει το αρχείο ανάγνωσης **src.txt** και ο **fpout**, για να ανοίξει το αρχείο εγγραφής **dest.txt**. Εάν υπάρξει σφάλμα στο άνοιγμα του αρχείου ανάγνωσης, το πρόγραμμα τερματίζεται (**exit(-1)**).

Με την πρόταση

```
cnt=fread(buf,sizeof(char),100,fpin);
```

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 τον buffer στο αρχείο εγγραφής **dest.txt** και ακολούθως επαναλαμβάνουμε την ανάγνωση.

Η πρόταση

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

γράφει στο **dest.txt** τα περιεχόμενα του **buffer** που μπορεί να παρέμειναν (εάν ο συνολικός αριθμός των δεδομένων δεν είναι ακριβές πολλαπλάσιο του μεγέθους του **buffer**).

Το πρόγραμμα ολοκληρώνεται με κλείσιμο των αρχείων εγγραφής και ανάγνωσης.

9.7 Ανάγνωση – εγγραφή γραμμή ανά γραμμή

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

```
char *fgets(char *pstr, int length, FILE *fp);
char *fputs(char *pstr, 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 έναν χαρακτήρα νέας γραμμής είτε αριθμό χαρακτήρων ίσο με **length-1**. Εάν η **fgets()** διαβάσει έναν 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 τον μηδενικό χαρακτήρα.

Στον ακόλουθο κώδικα

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

η πρόταση **fgets(buf,100,fpin)**:

- Θα διαβάσει ένα αλφαριθμητικό από το αρχείο που καθορίζει ο δείκτης **fpin** και θα το αποδώσει στον **buf**.
- Θα σταματήσει μετά τη *νέα γραμμή* ή τον **99^{ο}** χαρακτήρα.

- Θα τοποθετήσει ακολούθως στον **buf** τον μηδενικό χαρακτήρα.
- Θα επιστρέψει τον δείκτη του **buf** σε περίπτωση επιτυχίας ή **NULL** εάν ο **fpin** είναι άδειος.

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

9.7.1 Παράδειγμα

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

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

#define NAME "tape.txt"

int main()
{
  FILE *fp;
  char pchar[16];
  int i;
  float x[5];
  fp=fopen(NAME,"w");
  for (i=0;i<5;i++)
    fprintf( fp,"nm%d.dat\n",i+1 );
  fclose(fp);

  fp=fopen(NAME,"r");
  for (i=0;i<5;i++)
    printf( "line %d: %s\n",i+1,fgets(pchar,15,fp) );
  fclose(fp);

  fp=fopen("data.dat","w");
  for (i=0;i<5;i++)
    x[i]=sqrt(i);
  fwrite(x,sizeof(x),1,fp);
  fclose(fp);
  fp=fopen("data.dat","r");
  for (i=0;i<5;i++)
  {
    fscanf( fp,"%f",&x[i] );
    printf( "x[%d]=%f\n",i,x[i] );
  }
  fclose(fp);

  return 0;
}
```

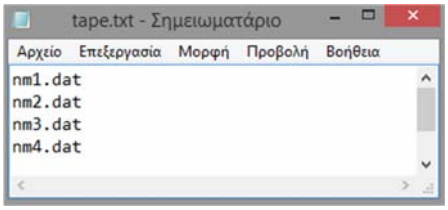
Αρχικά, ανοίγει το αρχείο κειμένου **tape.txt** για εγγραφή, στο οποίο εγγράφονται με χρήση της **fprintf()** πέντε αλφαριθμητικά που αποτελούν ονόματα αρχείων. Τα περιεχόμενα του **tape.txt** απεικονίζονται στην **Εικόνα 9.6.α**.

Μετά την εγγραφή των αλφαριθμητικών το αρχείο κλείνει και ανοίγει εκ νέου για ανάγνωση. Η ανάγνωση γίνεται με χρήση της **fgets()** και μετά το πέρας της το αρχείο κλείνει.

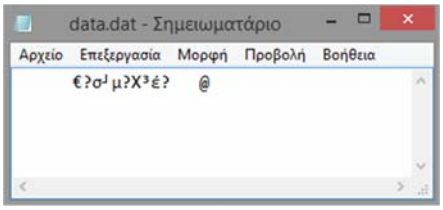
Ακολούθως το **tape.txt** ανοίγει για εγγραφή ως αρχείο κειμένου, γεγονός που σημαίνει ότι τα υφιστάμενα περιεχόμενα διαγράφονται. Η εγγραφή γίνεται με την πρόταση

```
fwrite(x,sizeof(x),1,fp);
```

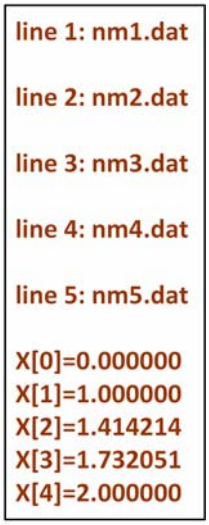
η οποία αντιγράφει ολόκληρο τον πίνακα **x** στο αρχείο. Στην **Εικόνα 9.6.β** απεικονίζονται τα περιεχόμενα του αρχείου μετά την εγγραφή, απ' όπου προκύπτει ότι τα δεδομένα είναι μη αναγνώσιμα από τον κειμενογράφο, γιατί είναι αποθηκευμένα ως δυαδικά δεδομένα. Συνεπώς, καθίσταται φανερό ότι, αν και το αρχείο **data.dat** άνοιξε ως αρχείο κειμένου, η χρήση της **fwrite()** το μετέτρεψε σε δυαδικό αρχείο. Ανοίγοντάς το εκ νέου για ανάγνωση είναι εφικτή η χρήση της **fscanf()**, μόνο εφόσον ζητηθούν επακριβώς ο αριθμός και ο τύπος των δεδομένων που είναι αποθηκευμένα, όπως φαίνεται και από τα αποτελέσματα της **Εικόνας 9.6.γ**.



Εικόνα 9.6.α *Τα περιεχόμενα του αρχείου κειμένου tape.txt του προγράμματος του παραδείγματος 9.7.1*



Εικόνα 9.6.β *Τα περιεχόμενα του δυαδικού αρχείου data.dat του προγράμματος του παραδείγματος 9.7.1*



Εικόνα 9.6.γ *Η έξοδος του προγράμματος του παραδείγματος 9.7.1*

9.8 Τυχαία προσπέλαση δυαδικού αρχείου

Στις προηγούμενες ενότητες 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 διαβαστεί 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 γλώσσα C παρέχει τη δυνατότητα **τυχαίας προσπέλασης** (random access), με την οποία παρέχεται πρόσβαση σε οποιοδήποτε 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–μέλος της δομής **FILE**. Όταν το αρχείο ανοίγει για ανάγνωση, η θέση αυτή είναι **0**. Όταν ανοίγει για προσάρτηση, είναι το τέλος του αρχείου. Καθορίζοντας τον δείκτη θέσης αρχείου μπορούμε να έχουμε προσπέλαση σε οποιοδήποτε σημείο του αρχείου.

9.8.1 Συναρτήσεις διαχείρισης της θέσης σε αρχείο

1. **`fseek()`**

Η συνάρτηση **`fseek()`** αποτελεί το εργαλείο για την εκτέλεση λειτουργιών τυχαίας ανάγνωσης και εγγραφής. Ορίζεται στο αρχείο κεφαλίδας **`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 όρισμα **`origin`**.
- **`origin`** είναι το σημείο αφετηρίας και μπορεί να έχει μία από τις τρεις ακόλουθες τιμές:
  - *(i)* *Αφετηρία*: αρχή του αρχείου, *τιμή*: **SEEK_SET(0)**
  - *(ii)* *Αφετηρία*: τρέχουσα θέση, *τιμή*: **SEEK_CUR(1)**
  - *(iii)* *Αφετηρία*: τέλος του αρχείου, *τιμή*: **SEEK_END(2)**

Έτσι, για να βρεθεί π.χ. το **`offset`** από την τρέχουσα θέση, 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);`**

2. **`ftell()`**

Η συνάρτηση **`ftell()`** ορίζεται στο αρχείο κεφαλίδας **`stdio.h`** και έχει το ακόλουθο πρωτότυπο:

```
long ftell(FILE *fptr)
```

όπου **`fptr`** είναι ένας δείκτης αρχείου, που επιστρέφεται από την **`fopen()`**. Επιστρέφει την τιμή ενός **`long`** ακεραίου, η οποία αντιστοιχεί στην τρέχουσα θέση στο αρχείο. Δηλαδή, η επιστρεφόμενη τιμή διαδραματίζει τον ρόλο του **`offset`** στην **`fseek()`**, όταν ως **`origin`** τεθεί η αρχή του αρχείου.

9.8.1.1 Παράδειγμα

Για τη διαχείριση των στοιχείων των φοιτητών ορίζεται ο πίνακας **`studentList[size]`** με στοιχεία τύπου δομής **`StudentT`**:

```
struct StudentT
{
  int AM,year;
  char firstname[20],lastname[40];
};
```

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

Για τη διαχείριση των στοιχείων των φοιτητούν θα αναπτυχθούν οι ακόλουθες συναρτήσεις:

1. **`saveData()`**: Αποθήκευση των δεδομένων του πίνακα **`studentList`** στο αρχείο **students.dat**.

2. **`readData()`**: Ανάγνωση των δεδομένων από το αρχείο και αποθήκευσή τους στον πίνακα **`studentList`**.

3. **`readStudent()`**: Προσπέλαση ενός συγκεκριμένου φοιτητή στο αρχείο και αποθήκευση των στοιχείων του σε μία μεταβλητή τύπου **`StudentT`**.

4. **`saveStudent()`**: 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 καθοριστεί με την εντολή προεπεξεργαστή **#define**.

1. saveData()

Η συνάρτηση καλείται από τη **main()** ως εξής:

```
saveData(fptr, studentList);
```

και έχει το ακόλουθο σώμα:

```
void saveData(FILE *fp, StudentT *plist)
{
  int i;
  rewind(fp);
  for (i=0;i<size;i++)
     fwrite(&plist[i],sizeof(StudentT),1,fp);
}
```

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

```
fwrite(plist,sizeof(StudentT),size,fp);
```

2. readData()

Η συνάρτηση καλείται από τη **main()** ως εξής:

```
readData(fptr, studentList);
```

και έχει το ακόλουθο σώμα:

```
void saveData( FILE *fp, StudentT *plist )
{
  rewind(fp);
  fread(plist,sizeof(StudentT),size,fp);
}
```

3. readStudent()

Η συνάρτηση καλείται από τη **main()** ως εξής:

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

όπου **svar** είναι μία οποιαδήποτε μεταβλητή τύπου **StudentT** (στη θέση της θα μπορούσε να είναι η **&studentList[i]**), **i** είναι η θέση ενός στοιχείου του πίνακα **studentList**. Το σώμα της είναι το ακόλουθο:

```
void read_student(FILE *fp, Student *ps, int k)
{
  fseek(fp,k*sizeof(StudentT),SEEK_SET);
  fread(ps,sizeof(StudentT),1,fp);
}
```

4. saveStudent()

Η συνάρτηση καλείται από τη **main()** ως εξής:

```
saveStudent(fptr, &studentList[i], i);
```

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

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

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

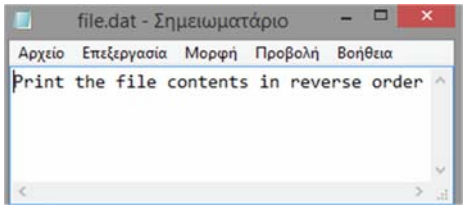
Η συνάρτηση **readStudent()** χρησιμοποιείται, για να αντλήσουμε τα στοιχεία του του φοιτητή από το αρχείο και πιθανώς να τα τροποποιήσουμε.

9.8.1.2 Παράδειγμα

Το πρόγραμμα που ακολουθεί, εμφανίζει τα περιεχόμενα ενός αρχείου με αντίστροφη σειρά.

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

int main(void)
{
   char ch;
   FILE *fp;
   long i,last;
   fp=fopen("file.dat","rb");      assert(fp!=NULL);
   fseek(fp,0,SEEK_END); /* μετακίνηση το τέλος του αρχείου */
   last=ftell(fp);
   for (i=1;i<=last;i++)
   {
      /* μετακίνηση από το τέλος προς τα πίσω */
      fseek(fp,-i,SEEK_END);
      ch=getc(fp);
      if (ch!=EOF)
         putchar(ch);
   }
   putchar('\n');
   fclose(fp);

   return 0;
}
```

Στην **Εικόνα 9.6.α** απεικονίζονται τα περιεχόμενα του αρχείου και στην **Εικόνα 9.6.β** η έξοδος του προγράμματος, από την οποία συνάγεται ότι εκτυπώθηκε αντεστραμμένη η πρόταση, που βρισκόταν αποθηκευμένη στο **file.dat**.



Εικόνα 9.7.α *Το αρχείο του προγράμματος του παραδείγματος 9.8.1.2*

Εικόνα 9.7.β *Η έξοδος του προγράμματος του παραδείγματος 9.8.1.2*

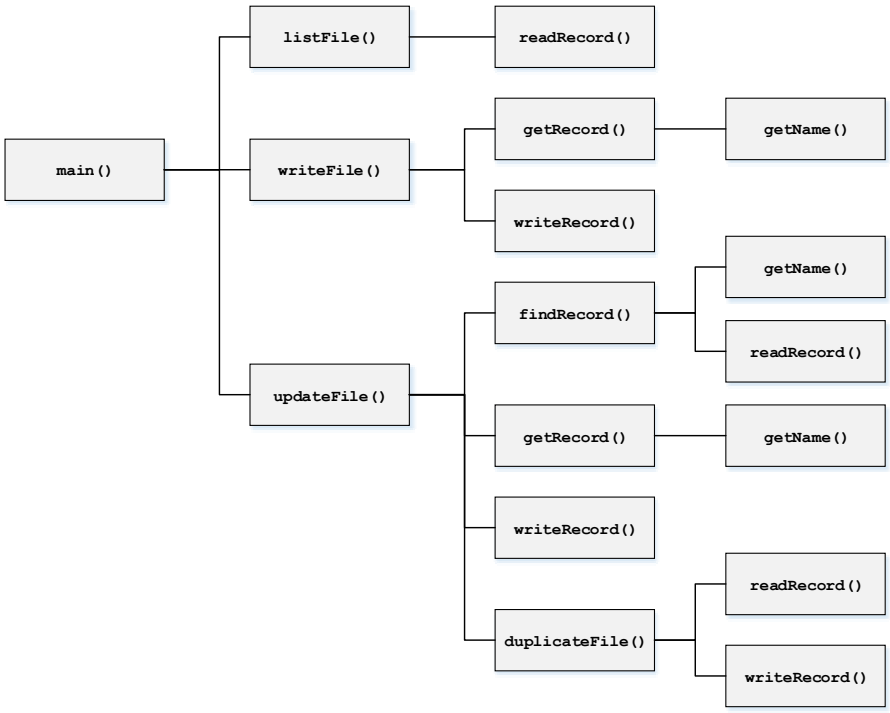
9.9 Παράδειγμα ανάπτυξης προγράμματος

Στο πρόγραμμα που ακολουθεί, θα χρησιμοποιηθούν σύνθετα δεδομένα τύπου δομής (εγγραφές), για να υλοποιηθούν ορισμένες διαδικασίες επεξεργασίας αρχείων, όπως η ενημέρωση των εγγραφών, η προσθήκη

και η διαγραφή εγγραφών. Στο πλαίσιο αυτό, οι λειτουργίες του προγράμματος θα μεριστούν σε δέκα συναρτήσεις:

- **`main()`**: Ελέγχει τη συνολική λειτουργία του προγράμματος και παρέχει στον χρήστη τη δυνατότητα επιλογής από ένα πλήθος λειτουργιών του αρχείου.
- **`updateFile()`**: Τροποποιεί τα περιεχόμενα του αρχείου.
- **`listFile()`**: Εμφανίζει τα περιεχόμενα του αρχείου στην οθόνη (**`stdout`**).
- **`writeFile()`**: Έχει διπλή λειτουργία: την εγγραφή σε ένα νέο αρχείο και την προσάρτηση σε ένα υφιστάμενο αρχείο.
- **`getRecord()`**: Λαμβάνει δεδομένα (εγγραφές) από το πληκτρολόγιο (**`stdin`**).
- **`getName()`**: Διαβάζει ένα όνομα από το πληκτρολόγιο.
- **`writeRecord()`**: Γράφει δεδομένα (εγγραφές) στο αρχείο.
- **`readRecord()`**:Λαμβάνει δεδομένα (εγγραφές) από το αρχείο.
- **`findRecord()`**: Αναζητά μία εγγραφή στο αρχείο, βάσει μίας μεταβλητής–μέλος της εγγραφής.
- **`duplicateFile()`**: Αναπαράγει το αρχείο αντικαθιστώντας μία εγγραφή, όταν αυτή έχει διαφορετικό μήκος από την αντικαθιστώμενη.

Η ιεραρχία κλήσεων των συναρτήσεων απεικονίζεται στο **Σχήμα 9.1**. Οι τρεις συναρτήσεις που καλούνται από τη **`main()`**, υλοποιούν την κύρια λειτουργικότητα του προγράμματος. Οι συναρτήσεις που βρίσκονται στα δεξιά τους απλοποιούν τις λειτουργίες των τριών κυρίων συναρτήσεων.



Σχήμα 9.1 *Η ιεραρχία των κλήσεων των συναρτήσεων του προγράμματος*

Τα δεδομένα του αρχείου (εγγραφές) είναι τύπου δομής. Χάριν ευκολίας θα χρησιμοποιηθεί μία στοιχειώδης δομή, αποτελούμενη από δύο μέλη:

```
struct RecordT
{
  char name[MAXLENGTH];
  int age;
```

```
};
```

1. Ανάγνωση εγγραφής από το πληκτρολόγιο

Το πρωτότυπο της συνάρτησης που θα λαμβάνει δεδομένα από το πληκτρολόγιο είναι το εξής:

```
struct RecordT *getRecord(struct RecordT *precord);
```

Η συνάρτηση έχει όρισμα δείκτη σε δομή **RecordT**, ο οποίος δείχνει σε δεδομένο και επιστρέφει τη διεύθυνσή του. Η συνάρτηση έχει το ακόλουθο σώμα:

```
struct RecordT *getRecord(struct RecordT *precord)
{
  if (!precord)
  {
    printf( "No Record object to store input." );
    return NULL;
  }
  printf( "\nEnter a name less than %d characters:", MAXLENGTH );
  getName(precord->name);
  printf( "Enter the age of %s: ", precord->name );
  scanf( " %d", &precord->age );
  return precord;
}
```

Η βοηθητική συνάρτηση **getName()** ορίζεται ως εξής:

```
void getNname(char *pname)  {
  int len;
  fflush(stdin);  /* Η fflush(stdin) αδειάζει τον χώρο προσωρινής
                     αποθήκευσης, για να γίνει σωστά η ανάγνωση */
  fgets(pname,MAXLENGTH,stdin);
  len=strlen(pname);
  if(pname[len-1]=='\n')   /* περίπτωση που υπάρχει αλλαγή γραμμής */
    pname[len-1] = '\0';
}
```

Επειδή θα χρειαστεί να αναγνωστούν δεδομένα σε διάφορα σημεία του προγράμματος, το ζήτημα της αλλαγής γραμμής το διαχειρίζεται ο ανωτέρω κώδικας. Εάν η είσοδος υπερβαίνει του **MAXLENGTH** χαρακτήρες, τότε η αλλαγή γραμμής θα παραμείνει στον χώρο προσωρινής αποθήκευσης και δεν θα αποθηκευτεί στον πίνακα που διαχειρίζεται ο **pname**. Η **getName()** διευθετεί αυτό το ζήτημα.

2. Αποθήκευση εγγραφής σε αρχείο

Το πρωτότυπο της συνάρτησης που θα αποθηκεύει εγγραφές στο αρχείο, είναι το εξής:

```
void writeRecord(struct RecordT *precord, FILE *pFile);
```

Το πρώτο όρισμα είναι δείκτης σε δομή τύπου **RecordT**, όπου θα περιέχονται τα προς εγγραφή δεδομένα. Η συνάρτηση έχει το ακόλουθο σώμα:

```
void writeRecord(struct RecordT *precord, FILE *pFile)
{
  if ((!precord) || (!pFile))
    printf("Error with the record or with the output file.");
  else
  {
    size_t length=strlen(precord->name);
    fwrite(&length,sizeof(length),1,pFile);
    fwrite(precord->name, sizeof(char),length,pFile);
    fwrite(&precord->age,sizeof(precord->age),1,pFile);
  }
```

```
}
```

Η συνάρτηση αρχικά εγγράφει το μήκος του αλφαριθμητικου και το ίδιο το αλφαριθμητικό χωρίς τον μηδενικό χαρακτήρα τερματισμού. Ο λόγος της απουσίας του μηδενικού χαρακτήρα είναι, για να επιτρέπεται ο κώδικας που θα διαβάσει το αρχείο να καθορίσει πόσοι χαρακτήρες υπάρχουν στο αλφαριθμητικό του ονόματος. Ακολούθως, εγγράφεται η τιμή της ηλικίας στο αρχείο.

3. Ανάγνωση εγγραφής από αρχείο

Το πρωτότυπο της συνάρτησης ανάγνωσης μίας εγγραφής από το αρχείο είναι το εξής:

```
struct RecordT *readRecord(struct RecordT *precord, FILE *pFile);
```

και έχει το ακόλουθο σώμα:

```
struct RecordT * readRecord(struct RecordT *precord, FILE *pFile)
{
  if ((!precord) || (!pFile))
  {
     printf("Error with the record or with the output file.");
     return NULL;
  }
  size_t length=0;
  fread(&length,sizeof(length),1,pFile);
  if (feof(pFile))
    return NULL;
  if (length+1>MAXLENGTH)
  {
     printf( "\nName too long. Exiting program." );
     exit(-1);
  }
  fread(precord->name,sizeof(char),length,pFile);
  precord->name[length]='\0'; /* προσαρτάται ο τερματιστής */
  fread(&precord->age,sizeof(precord->age),1,pFile);
  return precord;
}
```

Αρχικά, διαβάζεται το μήκος του ονόματος. Επειδή ο δείκτης θέσης του αρχείου μπορεί να βρίσκεται στο τέλος του, γίνεται έλεγχος με κλήση της `feof()`. Εάν ο δείκτης βρίσκεται στο τέλος, η συνάρτηση επιστρέφει το **NULL**, το οποίο σηματοδοτεί ότι ο δείκτης θέσης αρχείου βρίσκεται στο τέλος. Αντίστοιχα γίνεται έλεγχος για το μήκος του ονόματος: εάν υπερβεί το όριο το πρόγραμμα τερματίζεται, σε αντίθετη περίπτωση γίνεται η ανάγνωση της εγγραφής, προσαρτώντας τον μηδενικό χαρακτήρα στο τέλος του ονόματος.

4. Αποθήκευση σε αρχείο

Η συνάρτηση

```
void writeFile(char *filename, char *mode);
```

αποθηκεύει έναν αριθμό εγγραφών στο αρχείο. Το πρώτο όρισμα είναι το όνομα του αρχείου και το δεύτερο όρισμα είναι ο τρόπος εγγραφής. Εάν επιλεγεί το `"wb+"`, τότε εφόσον το αρχείο προϋπάρχει, θα γίνει προσάρτηση των νέων εγγραφών στο τέλος του.

Η συνάρτηση υλοποιείται ως εξής:

```
void writeFile(char *filename, char *mode)
{
  char answer='y';
  FILE *pFile;
  pFile=fopen(filename,mode);  assert(pFile!=NULL);
  do
  {
```

```
      struct RecordT record;
      writeRecord(getRecord(&record),pFile);
      printf( "Do you want to enter another(y or n)? " );
      scanf( "\n%c",&answer );
      fflush(stdin); /* αφαίρεση των λευκών διαστημάτων */
    } while(tolower(answer)=='y');
    fclose(pFile);
}
```

Ο βρόχος **do-while** επαναλαμβάνεται όσο ο χρήστης αποκρίνεται θετικά στο ερώτημα του προγράμματος για αποθήκευση νέας εγγραφής. Η χρήση της συνάρτησης **tolower()** γίνεται, για να μην υπάρχει διάκριση του **'y'** από το **'Y'**.

Η συνάρτηση χρησιμοποιεί τις προαναφερθείσες συναρτήσεις **getRecord()**,**writeRecord()** για την ανάγνωση και αποθήκευση της κάθε εγγραφής.

4. Εμφάνιση στην οθόνη του συνόλου των εγγραφών

Η συνάρτηση

```
void listFile(char *filename);
```

εμφανίζει στην οθόνη (προκαθορισμένη συσκευή εξόδου) όλες τις εγγραφές που υπάρχουν στο αρχείο και υλοποιείται ως εξής:

```
void listFile(char *filename)
{
  FILE *pFile;
  pFile=fopen(filename,"rb");
  assert(pFile!=NULL);
  struct RecordT record;
  printf( "\nThe contents of %s are:", filename );
  while(readRecord(&record,pFile)!= NULL)
    printf("\nName: %s, Age: %d\n",record.name,record.age );
  printf( "\n" );
  fclose(pFile);
}
```

Η συνάρτηση ανάγνωσης εγγραφής **readRecord()** καλείται μέσα σε έναν βρόχο **while**, έως ότου αναγνωσθεί και η τελευταία εγγραφή. Σε κάθε επανάληψη τα μέλη της εγγραφής που διαβάστηκε εμφανίζονται στην οθόνη με την **printf()**.

5. Μεταβολή/τροποποίηση εγγραφής

Η συνάρτηση

```
void updatefile(char *filename);
```

Μεταβάλλει μία εγγραφή ακόμη και στην περίπτωση που τα νέα περιεχόμενα δεν είναι ισομήκη με τα παλαιά. Το σώμα της είναι το ακόλουθο:

```
void updateFile(char *filename)
{
  char answer='y';
  FILE *pFile;
  pFile=fopen(filename,"rb+");
  assert(pFile!=NULL);
  struct RecordT record;
  int index=findRecord(&record,pFile);
  if (index<0)
    printf( "\nRecord not found." );
  else
```

```
   {
      printf( "\n%s is aged %d,",record.name,record.age );
      struct RecordT newrecord;
      printf( "\nYou can now enter the new name and age for
%s.",record.name );
      getRecord(&newrecord);
      if ((strlen(record.name)==strlen(newrecord.name)))
      {
         fseek(pFile,
-(long)(sizeof(size_t)+strlen(record.name)+sizeof(record.age)),
SEEK_CUR);
         writeRecord(&newrecord,pFile);
         fflush(pFile);
      }
      else
        duplicateFile(&newrecord,index,filename,pFile);
      printf( "File update complete.\n" );
   }
}
```

Η λειτουργία της συνάρτησης **updateFile()** περιγράφεται με μία σειρά απλών βημάτων:

(α) Άνοιγμα του αρχείου.

(β) Εύρεση της θέσης της εγγραφής (αριθμοδείκτη) που θα τροποποιηθεί, με την πρώτη εγγραφή να βρίσκεται στη θέση **0** (συνάρτηση **findRecord()**).

(γ) Ανάγνωση των δεδομένων της νέας εγγραφής με τη συνάρτηση **getRecord()**.

(δ) Έλεγχος εάν η εγγραφή που θα τροποποιηθεί είναι ισομήκης με τη νέα εγγραφή. Σε μία τέτοια περίπτωση, μετάβαση στη θέση της παλιάς εγγραφής με τη συνάρτηση **fseek()** και αντικατάστασή της από τη νέα με τη συνάρτηση **writeRecord()**. Σε αντίθετη περίπτωση, εκτέλεση της συνάρτησης **duplicateFile()**, η οποία αντιγράφει ολόκληρο το αρχείο σε ένα καινούριο, με τη νέα εγγραφή να έχει αντικαταστήσει την παλιά. Η συνάρτηση **duplicateFile()** έχει το ακόλουθο σώμα:

```
void duplicateFile(struct RecordT *pnewrecord, int index,
char *filename, FILE *pFile)
{
  int i;
  char tempname[L_tmpnam];
  if (tmpnam(tempname)==NULL)
  {
     printf( "\nTemporary file name creation failed." );
     exit(-1);
  }
  char tempfile[strlen(dirpath)+strlen(tempname)+1];
  strcpy(tempfile, dirpath);
  strcat(tempfile,tempname);
  FILE *ptempfile;
  ptempfile=fopen(tempfile,"wb+");
  struct RecordT record;
  for(i=0;i<index;i++)
     writeRecord(readRecord(&record,pFile),ptempfile);
  writeRecord(pnewrecord,ptempfile);
  readRecord(&record,pFile);
  while (readRecord(&record,pFile))
     writeRecord(&record, ptempfile);
  if (fclose(pFile)==EOF)
     printf( "\n Failed to close %s", filename );
```

```
      if (fclose(ptempfile)==EOF)
        printf( "\n Failed to close %s", tempfile );
      if (!remove(filename))
      {
        printf( "\nRemoving the old file failed. Check file in %s",
   dirpath );
        return;
      }
      if (!rename(tempfile,filename))
      printf( "\nRenaming the file copy failed. Check file in %s", dir-
 path );
    }
```

και η λειτουργία της περιγράφεται βηματικά ως εξής:
(α) Δημιουργία ενός νέου αρχείου στον ίδιο κατάλογο που βρίσκεται το υφιστάμενο αρχείο. Η μεταβλητή **dirpath** είναι καθολική και περιέχει την πλήρη διαδρομή μέσα στον σκληρό δίσκο.

(β) Αντιγραφή από το υφιστάμενο στο νέο αρχείο όλων των εγγραφών που προηγούνται της εγγραφής που θα μεταβληθεί.

(γ) Αποθήκευση της νέας εγγραφής στο καινούριο αρχείο και παράλειψη μεταφοράς της παλιάς εγγραφής σε αυτό.

(δ) Αντιγραφή από το υφιστάμενο στο νέο αρχείο των υπόλοιπων εγγραφών.

(ε) Κλείσιμο παλιού και νέου αρχείου.

(στ) Διαγραφή του παλιού αρχείου με τη συνάρτηση **remove()**, η οποία βρίσκεται στο αρχείο κεφαλίδας **stdio.h**.

(ζ) Μετονομασία του νέου αρχείου με τη συνάρτηση **rename()**, η οποία βρίσκεται στο αρχείο κεφαλίδας **stdio.h**. Το νέο αρχείο θα λάβει το όνομα του παλιού.

Η υλοποίηση της συνάρτησης **findRecord()**, η οποία καλείται από τη συνάρτηση **updateFile()** για να αναζητήσει τον αριθμοδείκτη της θέσης που βρίσκεται η αναζητηθείσα εγγραφής, έχει το ακόλουθο σώμα:

```
int findRecord(struct RecordT *precord, FILE *pFile)
{
  char name[MAXLENGTH];
  printf( "\nEnter the name for the record you wish to find: " );
  getName(name);
  rewind(pFile);
  int index=0;
  while(true)  /* boolean τιμή, με βάση το πρότυπο C99 */
  {
    readRecord(precord,pFile);
    if (feof(pFile))  return -1;
    if (!strcmp(name,precord->name))  break;
    ++index;
  }
  return index;
}
```

Η συνάρτηση διαβάζει από το πληκτρολόγιο το όνομα και το αναζητά στις εγγραφές του αρχείου. Εάν η αναζήτηση φτάσει στο τέλος του αρχείου χωρίς επιτυχία, επιστρέφεται το **-1** ως ένδειξη ότι δεν βρέθηκε η αναζητηθείσα εγγραφή. Εάν η αναζήτηση είναι επιτυχής, η συνάρτηση επιστρέφει τον αριθμοδείκτη της ευρεθείσας εγγραφής.

Ακολούθως, παρατίθεται το τελικό πρόγραμμα, όπου ο κώδικας των προαναφερθεισών συναρτήσεων παραλείπεται, καθώς παρουσιάστηκε προηγουμένως.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdbool.h> /* σύμφωνα με το πρότυπο C99 */

#define MAXLENGTH 80
const char *dirpath = "C:\\temp\\"; /* διαδρομή του καταλόγου στον
                                       οποίο βρίσκεται το αρχείο */
const char *file = "data.dat";      /* όνομα αρχείου */
struct RecordT
{
  char name[MAXLENGTH];
  int age;
};
/*-----------------------------------*/
void listFile(char *filename);
void updateFile(char *filename);
struct RecordT *getRecord(struct RecordT *precord);
void getName(char *pname);
void writeFile(char *filename, char *mode);
void writeRecord(struct RecordT *precord, FILE *pFile);
struct RecordT *readRecord(struct RecordT *precord, FILE *pFile);
int findRecord(struct RecordT *precord, FILE *pFile);
void duplicateFile(struct RecordT *pnewrecord, int index, char
*filename, FILE *pFile);
/*-----------------------------------*/
int main(void)
{
  char filename[strlen(dirpath)+strlen(file)+1];
  strcpy(filename, dirpath);
  strcat(filename, file);
  /* Επιλογή λειτουργίας */
  char answer='q';
  while(true)
  {
    printf("\nChoose one of the following options:"
    "\nTo list the file contents enter L"
    "\nTo create a new file enter C"
    "\nTo add new records enter A"
    "\nTo update existing records enter U"
    "\nTo delete the file enter D"
    "\nTo end the program enter Q\n : ");
    scanf("\n%c", &answer);
    switch(tolower(answer))
    {
      case 'l':
        listFile(filename);
      break;
      case 'c':
        writeFile(filename,"wb+");
        printf("\nFile creation complete.");
      break;
      case 'a':
```

```
                writeFile(filename, "ab+");
                printf("\nFile append complete.");
              break;
              case 'u':
                updateFile(filename);
              break;
              case 'd':
                printf("Are you sure you want to delete %s (y or n)? ",
filename);
                scanf("\n%c", &answer);
                if(tolower(answer)=='y')  remove(filename);
                   /* η συνάρτηση remove διαγράφει ένα αρχείο */
              break;
              case 'q': /* έξοδος από το πρόγραμμα */
                printf("\nEnding the program.", filename);
                return 0;
              default:
                printf("Invalid selection. Try again.");
              break;
            }
        }
        return 0;
      }
```

```
Choose one of the following options:
To list the file contents enter          L
To create a new file enter                       C
To add new records enter                         A
To update existing records enter         U
To delete the file enter                         D
To end the program enter                         Q
: c

Enter a name less than 30 characters:John Johnson
Enter the age of John Johnson: 35
Do you want to enter another(y or n)? y

Enter a name less than 30 characters:Jacob Jacobson
Enter the age of Jacob Jacobson: 72
Do you want to enter another(y or n)? n
File creation complete.

Choose from the following options:
To list the file contents enter          L
To create a new file enter                       C
To add new records enter                         A
To update existing records enter         U
To delete the file enter                         D
To end the program enter                         Q
: l

The contents of C:\temp\data.dat are:
John Johnson          Age: 235
Jacob Jacobson        Age: 72

Choose from the following options:
To list the file contents enter          L
To create a new file enter                       C
To add new records enter                         A
To update existing records enter         U
To delete the file enter                         D
To end the program enter                         Q
: u

Enter the name for the record you wish to find: Jacob Jacobson

Jacob Jacobson is aged 72,
You can now enter the new name and age for Jacob Jacobson.
Enter a name less than 30 characters:Jacob Powell
Enter the age of Kitty Moline: 72
File update complete.

Choose from the following options:
To list the file contents enter L
To create a new file enter C
To add new records enter A
To update existing records enter U
To delete the file enter D
To end the program enter Q
: q
```

Εικόνα 9.8 *Μία πιθανή έξοδος του προγράμματος*

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Ο αναγνώστης καλείται να επιλέξει μία από τις τέσσερις απαντήσεις.

(1) Στη συνάρτηση `fseek()`, με πρωτότυπο `int fseek(FILE *fptr, long offset, int origin)`, ποια από τις ακόλουθες τιμές δεν μπορεί να δοθεί στο όρισμα `origin`;

(α) SEEK_CUR
(β) SEEK_END
(γ) SEEK_START
(δ) SEEK_SET

(2) Ποιο από τα ακόλουθα σχόλια που αφορούν στην πρόταση `fgets(buf,100,pFin);` είναι λανθασμένο;

(α) Θα διαβάσει ένα αλφαριθμητικό από το αρχείο που καθορίζει ο δείκτης `pFin` και θα το αποδώσει στο `buf`.
(β) Θα σταματήσει μετά τη νέα-γραμμή `\n` ή τον 99^{ο} χαρακτήρα.
(γ) Εάν η `fgets()` διαβάσει έναν χαρακτήρα νέας γραμμής, ο τελευταίος θα αποτελέσει τμήμα του αλφαριθμητικού.
(δ) Θα επιστρέψει το -1 σε περίπτωση επιτυχίας ή **NULL**, εάν το αρχείο το οποίο διαχειρίζεται ο δείκτης `pFin` είναι κενό.

(3) Ο δείκτης θέσης αρχείου `fptr` επανατοποθετείται στην αρχή με την εντολή

(α) `reset(fptr);`
(β) `reset(fptr,0);`
(γ) `rewind(fptr);`
(δ) `rewind(fptr,0);`

(4) Ποια από τις ακόλουθες προτάσεις είναι λανθασμένη;

(α) Η γλώσσα C παρέχει τη δυνατότητα τυχαίας προσπέλασης (random access) αρχείου, με την οποία υπάρχει πρόσβαση σε οποιοδήποτε σημείο ενός αρχείου.
(β) Κάθε ανοικτό αρχείο έχει έναν δείκτη θέσης αρχείου, ο οποίος καθορίζει σε ποιο σημείο του αρχείου θα γίνει ανάγνωση ή εγγραφή και δίνεται ως αριθμός bytes από την αρχή του αρχείου.
(γ) Ο δείκτης θέσης αρχείου έχει τιμή -1, όταν το αρχείο ανοίγει για ανάγνωση.
(δ) Η τυχαία προσπέλαση αρχείου στηρίζεται στον καθορισμό του δείκτη θέσης αρχείου, έτσι ώστε να έχουμε προσπέλαση σε οποιοδήποτε σημείο του αρχείου.

(5) Ποια από τις διαπιστώσεις με βάση το ακόλουθο τμήμα κώδικα είναι λανθασμένη;

```
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);
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);
```

(α) Ορίζονται οι δείκτες αρχείου **pFin** και **pFout** και ο πίνακας χαρακτήρων 100 θέσεων **buffer**, εκ των οποίων ο **pFin** χρησιμοποιείται ,για να ανοίξει το αρχείο ανάγνωσης **src.txt**, και ο **pFout**, για να ανοίξει το αρχείο εγγραφής **dest.txt**.
(β) Με την πρόταση **cnt=fread(buf,sizeof(char),100,pFin);** ζητείται να αναγνωσθούν 100 χαρακτήρες από το **src.txt** και να αποθηκευτούν στον **buffer**.
(γ) Η συνθήκη **while** ελέγχει κατά πόσον γέμισε ο **buffer**, δηλαδή εάν γέμισε, γράφουμε όλα τα περιεχόμενα του **buffer** στο αρχείο εγγραφής **dest.txt** και περατώνουμε την ανάγνωση.
(δ) Η πρόταση **if (cnt!=0) fwrite(buf,sizeof(char),cnt,pFout);** γράφει στο **dest.txt** τα περιεχόμενα του **buffer**, που μπορεί να παρέμειναν.

Ασκήσεις

Άσκηση 1
Να γραφεί πρόγραμμα, το οποίο θα ενημερώνει θα μετατρέπει σε ένα προϋπάρχον αρχείο κειμένου όλα τα γράμματα του αγγλικού αλφαβήτου σε κεφαλαία.

Άσκηση 2
Να γραφεί πρόγραμμα, το οποίο θα διαβάζει ένα αρχείο κειμένου που θα περιέχει κώδικα στη γλώσσα C, θα αφαιρεί τα σχόλια και θα εγγράφει το υπόλοιπο περιεχόμενο σε ένα νέο αρχείο κειμένου.

Άσκηση 3
Να γραφεί πρόγραμμα, το οποίο θα διαβάζει τα περιεχόμενα ενός δυαδικού αρχείου input.dat, το οποίο περιέχει βαθμολογίες φοιτητών διαρθρωμένες σε ομάδες 40 βαθμών. Κάθε ομάδα 40 βαθμών αφορά στα μαθήματα του προγράμματος σπουδών ενός φοιτητή. Το πρόγραμμα θα υπολογίζει για κάθε φοιτητή τον μέσο όρο της βαθμολογίας του.

Άσκηση 4
Για τη διαχείριση των στοιχείων των συνδρομητών μίας εταιρείας, που παρέχει υπηρεσίες τηλεφωνίας, ορίζεται στη `main()` ο πίνακας `customerList[SIZE]` με στοιχεία τύπου δομής `CustomerT`, η οποία θα περιλαμβάνει το ονοματεπώνυμο του συνδρομητή, τη διεύθυνσή του (σε μεταβλητή τύπου δομής), το επάγγελμά του και τον τηλεφωνικό αριθμό του. Για τη διαχείριση μεμονωμένων συνδρομητών, μέσα στη `main()` ορίζεται η μεταβλητή `svar`, επίσης τύπου δομής `CustomerT`. Ο αριθμός των συνδρομητών `SIZE` καθορίζεται με εντολή προεπεξεργαστή `#define`.
Ζητείται:
***(α)* Να οριστεί ο τύπος δομής `CustomerT`.**
***(β)* Να γραφούν συναρτήσεις που να επιτελούν τα ακόλουθα:**
- **`void saveData(FILE *fp, CustomerT *plist)`: Αποθήκευση των δεδομένων του πίνακα `customerList` στο δυαδικό αρχείο customers.dat.**
- **`void readData(FILE *fp, CustomerT *plist)`: Ανάγνωση των δεδομένων από το αρχείο και αποθήκευσή τους στον πίνακα `customerList`.**
- **`void readCustomer(FILE *fp, CustomerT *ps, int k)`: Προσπέλαση ενός συνδρομητή, που βρίσκεται στην `k` θέση του αρχείου, και αποθήκευση των στοιχείων του μέσω του δείκτη `ps` στον χώρο μνήμης της μεταβλητής `svar` της `main()` .**
- **`void saveCustomer(FILE *fp, CustomerT *ps, int k)`: Αποθήκευση στο αρχείο των στοιχείων ενός συνδρομητή που βρίσκεται στην `k` θέση του πίνακα `customerList`.**

Η συνάρτηση `readCustomer()` χρησιμοποιείται, για να καταστούν διαθέσιμα τα στοιχεία του *k*-στου συνδρομητή από το αρχείο. Η συνάρτηση `saveCustomer()` χρησιμοποιείται, όταν έχουμε κάνει αλλαγές στα στοιχεία ενός συνδρομητή και θέλουμε να ενημερώσουμε την εγγραφή του στο αρχείο.
***(γ)* Να γραφεί τμήμα της `main()`, που θα περιλαμβάνει μόνο δήλωση των κατάλληλων μεταβλητών και από μία κλήση στις ανωτέρω συναρτήσεις.**
<u>Παρατήρηση</u>: Απαιτείται η χρήση εργαλείων για τυχαία προσπέλαση δυαδικού αρχείου.

Άσκηση 5

Να γραφεί πρόγραμμα, το οποίο:

(α) **Θα διαβάζει έως το τέλος του το προϋπάρχον δυαδικό αρχείο input_file.txt, στο οποίο βρίσκονται αποθηκευμένοι ακέραιοι αριθμοί, και θα υπολογίζει το πλήθος τους, το οποίο θα αποθηκεύει στην ακέραια μεταβλητή `size`. (*Υπόδειξη:* Για να διαβαστούν τα δεδομένα ένα προς ένα έως το τέλος του αρχείου, χωρίς να είναι γνωστός εκ των προτέρων ο αριθμός τους, μπορεί να χρησιμοποιηθεί ένας πίνακας μίας θέσης, π.χ. `arr[1]`, ως προσωρινός χώρος αποθήκευσης του δεδομένου σε κάθε κλήση της `fread()`. Να ληφθεί υπόψη ότι η `fread()` επιστρέφει έναν ακέραιο, που ισούται με τον αριθμό των δεδομένων που ανεγνώσθησαν σε κάθε κλήση της, ανεξάρτητα του αριθμού των δεδομένων που ζητήθηκε να αναγνωσθούν).**

(β) **Θα δεσμεύει μνήμη για `size` ακέραιους αριθμούς με χρήση της συνάρτησης `malloc()`. Τη μνήμη θα διαχειρίζεται ο δείκτης σε ακέραιο με όνομα `pArr`.**

(γ) **Θα διαβάζει εκ νέου το αρχείο `input_file.txt`, αποδίδοντας τους ακεραίους που περιέχει στον πίνακα `pArr`.**

(δ) **Στη συνέχεια θα καλείται μέσα από τη `main()` η συνάρτηση `void pwr(int *parray, int arraySize)`, η οποία θα μεταβάλλει τις τιμές των δεδομένων που διαχειρίζεται ο δείκτης `parray`, υψώνοντας στο τετράγωνο κάθε δεδομένο (κλήση της συνάρτησης μέσα στη `main()`: `pwr(pArr, size);`).**

(ε) **Η `main()` θα τελειώνει με την εγγραφή στο δυαδικό αρχείο output_file.txt των νέων τιμών των στοιχείων του πίνακα `pArr`, και την απελευθέρωση της δεσμευθείσας μνήμης με χρήση της συνάρτησης `free()`.**

Άσκηση 6

Να γραφεί πρόγραμμα το οποίο θα επιτελεί τα ακόλουθα:

(α) **Θα ανοίγει: *(i)* για ανάγνωση προϋπάρχον δυαδικό αρχείο finput.dat, στο οποίο θα βρίσκονται αποθηκευμένοι αριθμοί κινητής υποδιαστολής, *(ii)* για εγγραφή αρχείο κειμένου foutput.dat.**

(β) **Θα καλείται η συνάρτηση `int transferData(file *finput, file *foutput)`, η οποία θα διαβάζει τα δεδομένα από το αρχείο finput.dat και θα τα εγγράφει στο αρχείο foutput.dat, στη μορφή τριών ακεραίων ανά γραμμή. Ο αριθμός των συμπληρωμένων τριάδων θα επιστρέφει στη `main()`, ενώ τα δεδομένα που δεν συμπληρώνουν τριάδες (τελευταίο ή τελευταίο και προτελευταίο), δεν θα εγγράφονται στο αρχείο foutput.dat.**

(γ) **Θα καλείται η συνάρτηση `float meanRow(file *foutput, int n)`, η οποία θα διαβάζει από το αρχείο κειμένου τα δεδομένα (`n` είναι ο αριθμός των γραμμών–τριάδων) και θα επιστρέφει στη `main()` τη μέση τιμή των μεγίστων κάθε γραμμής.**

Άσκηση 7

Να τροποποιηθεί η Άσκηση 7 του κεφαλαίου 8 ως προς τις συναρτήσεις:

- **`void readCircle(circleT *pc, FILE *f1)`, η οποία θα διαβάζει από το αρχείο κειμένου, που προσπελαύνεται μέσω του δείκτη σε αρχείο `f1`, τιμές για τα μέλη της μεταβλητής στην οποία δείχνει ο δείκτης `pc`.**
- **`void printCircle(circleT *pc, FILE *f1)`, η οποία θα εμφανίζει στην οθόνη και θα τυπώνει στο αρχείο κειμένου, που προσπελαύνεται μέσω του δείκτη σε αρχείο `f1`, τα περιεχόμενο της μεταβλητής στην οποία δείχνει ο δείκτης `pc`.**
- **Μέσα στη συνάρτηση `main()` θα δημιουργούνται οι δείκτες αρχείου κειμένου `pf1` (για το αρχείο ανάγνωσης input_file.txt) και `pf2` (για το αρχείο εγγραφής output_file.txt). Πλέον, οι συναρτήσεις `readCircle()` και `printCircle()` θα καλούνται με ορίσματα `readCircle(&cir,pf1)` και `printCircle(&cir,pf2)`, αντίστοιχα. Το αρχείο input_file.txt θεωρείται πως προϋπάρχει στον κατάλογο c:\temp\. Στον ίδιο κατάλογο θα τοποθετηθεί το αρχείο output_file.txt.**

Βιβλιογραφία κεφαλαίου

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

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

Horton, I. (2006), *Beginning C – from Novice to Professional, 4th ed.*, Apress.

Prata, S. (2014), *C Primer Plus*, 6th ed., Addison-Wesley.

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

10. Γραμμικές δομές δεδομένων

Σύνοψη
Στο κεφάλαιο αυτό μελετώνται διάφορες δομές δεδομένων. Αρχικά παρουσιάζονται οι κατηγορίες και τα χαρακτηριστικά των δομών δεδομένων και ακολούθως μελετώνται οι σειριακές δομές της στοίβας και της ουράς, καθώς και εφαρμογές τους. Στη συνέχεια, εισάγεται η έννοια της συνδεδεμένης λίστας και μελετώνται η απλά και η διπλά συνδεδεμένη λίστα, όπως και η κυκλική λίστα. Στην επόμενη ενότητα αναπτύσσονται οι υλοποιήσεις της στοίβας και της ουράς ως απλά συνδεδεμένες λίστες. Το κεφάλαιο ολοκληρώνεται με ένα εκτενές παράδειγμα ανάπτυξης προγράμματος.

Λέξεις κλειδιά
γραμμικές/στατικές/δυναμικές δομές δεδομένων – κόμβος – στοίβα – ώθηση – εξώθηση – ουρά – εισαγωγή – εξαγωγή – απλά/διπλά συνδεδεμένη λίστα – κυκλική λίστα – δείκτης κεφαλής/ουράς.

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C – μεταβλητές – εκφράσεις – τελεστές – έλεγχος ροής προγράμματος – συναρτήσεις – πίνακες – δείκτες – δυναμική διαχείριση μνήμης – δομές – αρχεία

10.1 Γενικά

Στα προηγούμενα κεφάλαια χρησιμοποιήθηκε ο πίνακας ως συλλογή ομοειδών μεταβλητών. Ο πίνακας αποτελεί μία **δομή δεδομένων** (data structure) και μάλιστα **γραμμική** (linear), καθώς του *i*-στου στοιχείου του πίνακα προηγείται ένα μόνο στοιχείο (το στοιχείο *i*-1) και έπεται ένα μόνο στοιχείο (το στοιχείο *i*+1). Εάν σε κάποια δομή δεδομένων τα στοιχεία της συνδέονται με περισσότερα του ενός επόμενα ή/και με περισσότερα του ενός προηγούμενα, τότε η δομή ονομάζεται **μη γραμμική** (nonlinear). Γενικά, μία δομή δεδομένων αναπαριστά οντότητες του φυσικού κόσμου μέσω ενός αφηρημένου μοντέλου, στο οποίο προσδιορίζονται λειτουργίες (πράξεις). Το μοντέλο αυτό καλείται **αφηρημένος τύπος δεδομένων** (abstract data type) (ΑΤΔ).

Σε ό,τι αφορά τον τρόπο δημιουργίας, οι δομές δεδομένων διακρίνονται στις: *(α)* **στατικές**, όπως ο πίνακας, στις οποίες το μέγεθος της μνήμης καθορίζεται από την αρχή στο πρόγραμμα και όχι κατά τη διάρκεια της εκτέλεσής του και *(β)* στις **δυναμικές**, στις οποίες το μέγεθός τους μεταβάλλεται ανάλογα με τα δεδομένα που εισάγονται ή διαγράφονται κατά τον χρόνο εκτέλεσης του προγράμματος.

Ο προσδιορισμός της θέσης ενός στοιχείου γίνεται είτε με χρήση αριθμοδείκτη (index), όπως στην περίπτωση των πινάκων, όπου τα δεδομένα είναι σειριακά αποθηκευμένα, είτε με χρήση δείκτη (pointer), ο οποίος χρησιμοποιείται για να συνδέει δύο διαδοχικά στοιχεία σε μία δομή δεδομένων, όταν αυτά δεν είναι σειριακά αποθηκευμένα. Η δομή της πρώτης περίπτωσης ονομάζεται **σειριακή γραμμική λίστα** (sequential linear list) και η δομή της δεύτερης περίπτωσης καλείται **συνδεδεμένη γραμμική λίστα** (linked linear list).

Η βασική μονάδα των δομών δεδομένων – το «στοιχείο» της δομής δεδομένων – είναι ο **κόμβος** (node). Έως τώρα στους πίνακες ο κόμβος ταυτιζόταν με την τιμή ενός δεδομένου, καθώς το όνομα του πίνακα παρείχε το πρώτο byte του χώρου αποθήκευσης του πίνακα και ο αριθμοδείκτης του στοιχείου, που είχε ως τιμή το συγκεκριμένο δεδομένο, πληροφορούσε για το προηγούμενο και το επόμενο στοιχείο του πίνακα. Ωστόσο, στη γενική περίπτωση ένας κόμβος αποτελείται από δύο τμήματα: *(α)* το τμήμα της πληροφορίας ή δεδομένου (data), στο οποίο φιλοξενείται η τιμή του δεδομένου και *(β)* το τμήμα της διεύθυνσης (next), στο οποίο φιλοξενείται η διεύθυνση του επόμενου κόμβου με τον οποίο συνδέεται ο παρών. Η μορφή του κόμβου απεικονίζεται στο **Σχήμα 10.1**:

Σχήμα 10.1 *Απεικόνιση του κόμβου*

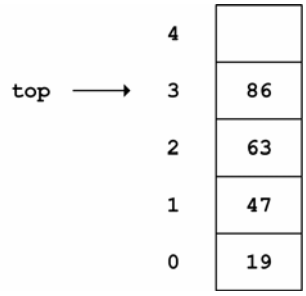
Οι βασικές λειτουργίες που επιτελούνται σε μία δομή δεδομένων, είναι οι ακόλουθες:

- *Προσπέλαση κόμβου* (access): Πρόσβαση σε κόμβο της δομής με σκοπό τη χρήση ή την τροποποίηση του περιεχομένου του.
- *Εισαγωγή κόμβου* (insertion): Προσθήκη νέου κόμβου σε υφιστάμενη δομή ή δημιουργία νέας δομής με την εισαγωγή του πρώτου κόμβου.
- *Διαγραφή κόμβου* (deletion): Αφαίρεση κόμβου από δομή.
- *Αναζήτηση* (searching): Εύρεση ενός ή περισσότερων κόμβων που περιέχουν συγκεκριμένη πληροφορία, η οποία αποτελεί το κλειδί της αναζήτησης.
- *Μεταβολή /τροποποίηση* (modification) του περιεχομένου ενός κόμβου.
- *Προσάρτηση* (append): Εισαγωγή κόμβου στο τέλος της δομής. Ο νέος κόμβος είναι ίδιας μορφής με εκείνους της δομής.
- *Ταξινόμηση* (sorting): Διάταξη των κόμβων μίας δομής κατά αύξουσα ή φθίνουσα σειρά.
- *Συγχώνευση* (merging): Συνένωση δύο ή περισσότερων δομών σε μία δομή, βάσει κανόνων τοποθέτησης των κόμβων στη νέα δομή.
- *Διαχωρισμός* (separation): Διάσπαση μίας δομής σε δύο ή περισσότερες.

10.2 Στοίβα

Η στοίβα (stack) είναι μία μορφή οργάνωσης των δεδομένων, στην οποία το δεδομένο που τοποθετείται τελευταίο στη στοίβα, ανασύρεται πρώτο. Μπορούμε να την παραλληλίσουμε με μία στοίβα από πιάτα, όπου κάθε νέο πιάτο τοποθετείται στην κορυφή (top) και χρησιμοποιείται πρώτο. Αυτή η μέθοδος επεξεργασίας ονομάζεται **LIFO** (**L**ast **I**n **F**irst **O**ut).

Εάν η στοίβα υλοποιηθεί με πίνακα (στατική στοίβα), απεικονίζεται ως εξής (αριστερά των κελιών υπάρχει η αρίθμησή τους):



Σχήμα 10.2 *Απεικόνιση στοίβας*

Ο **top** είναι ένας δείκτης που δείχνει στην κορυφή της στοίβας. Από την απεικόνιση της στοίβας στο **Σχήμα 10.2** προκύπτει ότι βρίσκονται αποθηκευμένα δεδομένα στις πρώτες **4** θέσεις, άρα ο **top** δείχνει στην τέταρτη θέση, δηλαδή θα αποκτήσει την τιμή **3** (η αρίθμηση των θέσεων ξεκινά από το **0**). Για να προσπελαστεί το δεδομένο **47**, πρέπει πρώτα να αφαιρεθούν τα δεδομένα **86** και **63**, ώστε το **47** να βρεθεί στην κορυφή της στοίβας. Αντίστοιχα, εάν στα υπάρχοντα τέσσερα δεδομένα προστεθεί ένα νέο δεδομένο, αυτό θα αποθηκευτεί στην $5^{η}$ θέση, δηλαδή ο **top** θα μετακινηθεί κατά μία θέση προς τα πάνω.

Θα πρέπει να σημειωθεί ότι στην υλοποίηση της στοίβας με πίνακα, ο πίνακας αποτελεί μόνο το μέσο αποθήκευσης. Η λειτουργία της στοίβας διαφέρει από εκείνη ενός κλασικού πίνακα, καθώς αφενός μεν δεν έχει σταθερό πλήθος στοιχείων, αφετέρου δε μόνο το δεδομένο που βρίσκεται στην κορυφή της στοίβας μπορεί να προσπελαστεί.

Δύο είναι οι κύριες λειτουργίες στη στοίβα:

- **Ώθηση** (*push*) στοιχείου στην κορυφή της στοίβας.
- **Απώθηση** (*pop*) στοιχείου από τη στοίβα.

Η διαδικασία της ***ώθησης*** πρέπει οπωσδήποτε να ελέγχει μήπως η στοίβα είναι γεμάτη, οπότε έχουμε υπερχείλιση (*overflow*).

Αντίστοιχα, η διαδικασία της ***απώθησης*** πρέπει να ελέγχει αν η στοίβα έχει αδειάσει, οπότε έχουμε υποχείλιση (*underflow*).

Η υλοποίηση των δύο λειτουργιών δίνεται ακολούθως, όπου χάριν ευκολίας θεωρείται ότι τα δεδομένα είναι τύπου ακεραίου:

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

#define N 100          /* Το μέγεθος του πίνακα για την αποθήκευση της
                          στοίβας */

void push(int stack[], int *t, int obj);
int pop(int stack[], int *t);
/*-------------------------------------*/
int main()
{
  int stack[N];
  int getStackValue;      /* Μεταβλητή υποδοχής των δεδομένων μετά
                             από pop */
  int top=-1;             /* Αρχικοποίηση του δείκτη, ώστε με την
                             έναρξη εισαγωγής δεδομένων ο δείκτης να
                             λάβει την τιμή 0  */
  ............
  push(stack,&top,<κάποιος ακέραιος>);
  ............
  push(stack,&top,<κάποιος ακέραιος>);
  ............
  getStackValue==pop(stack,&top);
  ............
  return 0;
}
/*-------------------------------------*/
void push(int stack[], int *t, int obj)
{
  if ((*t)==(N-1))
  {
     printf( "ERROR! Stack overflow...\n" );
     getchar();
     abort();
  }
  else  stack[++(*t)] = obj;
}
/*-------------------------------------*/
int pop(int stack[], int *t)
{
  int r;
  if ((*t)<0)
  {
     printf( "ERROR! Empty stack, unable to pop...\n" );
     getchar();
```

```
            abort();
        }
        else
            r=stack[(*t)--];
        return(r);
    }
```

10.2.1 Παράδειγμα

Το πρόγραμμα που ακολουθεί, υλοποιεί τον έλεγχο σωστής χρήσης των παρενθέσεων στις αριθμητικές εκφράσεις. Για παράδειγμα, η παράσταση **(a*(b+c)+d)** χρησιμοποιεί σωστά τις παρενθέσεις, ενώ οι παραστάσεις **(a*b+c)+d)**, **(a-b))*b+c)** δεν χρησιμοποιούν σωστά τις παρενθέσεις, γιατί πλεονάζουν δεξιές παρενθέσεις. Αντίστοιχα, η παράσταση **(a*(b+c+d)** δεν τις χρησιμοποιεί σωστά, γιατί πλεονάζει μία αριστερή παρένθεση.

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

#define N 10

void push(char stack[], int *t, char obj);
/*Εφόσον τα στοιχεία που αποσύρονται από τη στοίβα δεν χρησιμοποιούνται, η συνάρτηση pop δεν χρειάζεται να έχει επιστρεφόμενη τιμή */
void pop(char stack[], int *t);

int main()
{
  char stack[N],expr[30];
  int i,error,top=-1;

  printf( "Input expression, ENTER to quit:\n" );
  gets(expr);
  error=0;
  i=0;
  while ((expr[i]!='\0') && (error==0))
  {
     switch (expr[i])
     {
        case '(':
          push(stack,&top,expr[i]);
        break;
        case ')':
          if (top==-1)
             error=1;
          else
             pop(stack,&top);
        break;
     }
     i++;
  }
  if (top==-1)
     if (error==0)
        printf( "Correct expression." );
     else
```

```
        printf( "ERROR! Too many right parentheses." );
    else
      printf( "ERROR. Too many left parentheses." );

    return 0;
}
/*-----------------------------------*/
void push(char stack[], int *t, char obj)
{
  if ((*t)==(N-1))
  {
    printf( "ERROR! Stack overflow...\n" );
    getchar();
    abort();
  }
  else stack[++(*t)] = obj;
}
/*-----------------------------------*/
void pop(char stack[], int *t)
{
  char r;
  if ((*t)<0)
  {
    printf( "ERROR! Empty stack, unable to pop...\n" );
    getchar();
    abort();
  }
  else
    r=stack[(*t)--];
}
```

(α) *(β)* *(γ)*

Εικόνα 10.1 *Οι τρεις περιπτώσεις αποτελεσμάτων του προγράμματος του παραδείγματος 10.2.1*

10.2.2 Παράδειγμα

Μία κλασική εφαρμογή που χρησιμοποιεί τη δομή της στοίβας, είναι η επεξεργασία από τις γλώσσες προγραμματισμού αριθμητικών εκφράσεων, όπως **2+3 ή 2*(3+4)** ή ακόμα **((2+4)*7)+3*(9-5)**. Αυτή η μορφή παράστασης ονομάζεται *ένθετη* (infix) και έχει το χαρακτηριστικό ότι οι τελεστές τοποθετούνται μεταξύ των τελεσταίων (operands).

Κάθε ένθετη παράσταση χρησιμοποιεί προτεραιότητες στη σειρά εκτέλεσης των πράξεων. Π.χ. στην παράσταση **3+4*5** έχει προτεραιότητα η πράξη του πολλαπλασιασμού. Βέβαια, οι προτεραιότητες τροποποιούνται με τη χρήση παρενθέσεων, π.χ. **2*(3+4)** προτεραιότητα έχει η πράξη μέσα στις παρενθέσεις, δηλαδή η πρόσθεση. Οι προτεραιότητες των τελεστών παρατίθενται στον **Πίνακα 2.7**.

Εάν, όμως, ο μεταφραστής προσπαθήσει να εκτελέσει μία παράσταση στη ένθετη μορφή της, όπως π.χ. την παράσταση **3+4*5**, όταν φτάσει στον τελεστή της πρόσθεσης, δεν γνωρίζει εάν θα πρέπει να εκτελέσει την πρόσθεση ή να την αναβάλει για αργότερα, γιατί πιθανόν να προηγείται κάποια άλλη πράξη. Για τον λόγο αυτόν ο μεταφραστής ακολουθεί τις ακόλουθες δύο διαδικασίες για τον σωστό υπολογισμό των εκφράσεων:

- Η ένθετη μορφή της αριθμητικής έκφρασης μεταφράζεται σε επιθεματική (postfix) μορφή, όπου οι τελεστές εμφανίζονται μετά τους τελεσταίους. Π.χ. η παράσταση **2+3** γίνεται **23+** και η παράσταση **3+4*5** μετατρέπεται σε **345*+**.
- Η επιθεματική μορφή χρησιμοποιείται, κατόπιν, για τον υπολογισμό της έκφρασης. Η επιθεματική μορφή ονομάζεται και *Αντίστροφη Πολωνική Σημειογραφία* (Reverse Polish Notation – RPN).

Για τη μετατροπή μιας ένθετης μορφής σε επιθεματική ακολουθείται μία διαδικασία σάρωσης της αριθμητικής έκφρασης και εφαρμογής των παρακάτω κανόνων:

1. Αν το στοιχείο είναι τελεσταίος, τότε τοποθετείται στα δεξιά της επιθεματικής μορφής.
2. Αν το στοιχείο είναι τελεστής, τότε κατευθύνεται σε μία στοίβα. Εκεί συγκρίνεται με τον τελεστή της κορυφής της στοίβας.
3. Αν ο εισερχόμενος τελεστής έχει μεγαλύτερη προτεραιότητα από τον τελεστή της κορυφής της στοίβας, τότε ο τελεστής τοποθετείται στη στοίβα (*push*).
4. Αν η προτεραιότητα του εισερχόμενου τελεστή είναι μικρότερη ή ίση από την προτεραιότητα του τελεστή της κορυφής της στοίβας, τότε:
   - Εξάγονται όλοι οι τελεστές της στοίβας με προτεραιότητα μεγαλύτερη ή ίση από την προτεραιότητα του νέου τελεστή (*pop*) και τοποθετούνται στα δεξιά της επιθεματικής μορφής.
   - Ο νέος τελεστής τοποθετείται στη στοίβα (*push*).

Όταν η αριθμητική έκφραση έχει παρενθέσεις, τότε ακολουθούνται οι παρακάτω κανόνες:

1. Η αριστερή παρένθεση τοποθετείται αμέσως στη στοίβα (*push*).
2. Η δεξιά παρένθεση προκαλεί την εξαγωγή (*pop*) όλων των τελεστών, μέχρι να συναντηθεί η αριστερή παρένθεση στη στοίβα. Ύστερα, οι δύο παρενθέσεις αγνοούνται.

Για παράδειγμα, η παράσταση **(2*(3/2+4)-3)/2** μετατρέπεται σε **232/4+*3-2/** σύμφωνα με τα ακόλουθα βήματα:

Βήμα	**Σάρωση ενθεματικής παράστασης**	**Στοίβα**	**Επιθεματική μορφή**
1	**(**	**(**	
2	**2**	**(**	**2**
3	*****	**(***	**2**
4	**(**	**(*(**	**2**
5	**3**	**(*(**	**23**
6	**/**	**(*(/**	**23**
7	**2**	**(*(/**	**232**
8	**+**	**(*(/+**	**232/**
9	**4**	**(*(/+**	**232/4**
10	**)**	**(***	**232/4+**
11	**-**	**(-**	**232/4+***
12	**3**	**(-**	**232/4+*3**
13	**)**		**232/4+*3-**
14	**/**	**/**	**232/4+*3-**
15	**2**	**/**	**232/4+*3-2**
16	**τέλος**		**232/4+*3-2/**

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

#define N 100

void push(char stack[],int *t, char obj);
char pop(char stack[],int *t);
void gotoper(char stack[], int *t, char charVar, int intVar, char
postfix[], int *k);
void gotparen(char stack[], int *t, char postfix[], int *k);
```

```
int main()
{
  char stack[N];
  int i,j,top=-1;
  char infix[100],postfix[100];

  printf( "Input expression, ENTER to quit:\n" );
  gets(infix);
  i=0;
  j=0;
  while (infix[i]!='\0')
  {
     switch (infix[i])
     {
        case '+':
        case '-':
          gotoper(stack,&top,infix[i],3,postfix,&j);
        break;
        case '*':
        case '/':
          gotoper(stack,&top,infix[i],4,postfix,&j);
        break;
        case '(':
          push(stack,&top,infix[i]);
        break;
        case ')':
          gotparen(stack,&top,postfix,&j);
        break;
        default:
          postfix[j++]=infix[i];
        break;
     }
     i++;
  }
  postfix[j]='\0';
  j=0;
  while (postfix[j]!='\0')
  {
     printf( "%c",postfix[j] );
     j++;
  }

  return 0;
}
/*-------------------------------------*/
/*Για λόγους εξοικονόμησης χώρου, τα σώματα των συναρτήσεων push/pop
παραλείπονται, καθώς ταυτίζονται με εκείνα του Παραδείγματος 10.2.1   */
/*-------------------------------------*/
void gotoper(char stack[], int *t, char charVar, int intVar, char
postfix[], int *k)
{
  int top,prectop,done=0;
  char optop;
  top=*t;
  while((top!=-1) && (done==0))
```

```
   {
      if (stack[top]=='(')
      {
         prectop=1;
         done=1;
      }
      else
      {
         if ((stack[top]=='+') || (stack[top]=='-'))
            prectop=3;
         else
            prectop=4;
         if (prectop<intVar)
            done=1;
         else
            postfix[(*k)++]=pop(stack,&top);
      }
   }
   push(stack,&top,charVar);
   *t=top;
}
/*-------------------------------------*/
void gotparen(char stack[], int *t, char postfix[], int *k)
{
   int top,done=0;
   char cs;
   top=*t;
   while ((top!=-1) && (done==0))
   {
      cs=pop(stack,&top);
      if (cs=='(')
         done=1;
      else
         postfix[(*k)++]=cs;
   }
   *t=top;
}
```

10.3 Ουρά

Την έννοια της ουράς τη συναντάμε συχνά στην καθημερινή μας ζωή, π.χ. ουρά αναμονής με ανθρώπους. Το άτομο που είναι πρώτο στην ουρά, εξυπηρετείται και εξέρχεται. Το άτομο που μόλις καταφτάνει, τοποθετείται στο τέλος της ουράς. Αυτή η μέθοδος αυτή επεξεργασίας ονομάζεται **FIFO** (**F**irst **I**n **F**irst **O**ut).

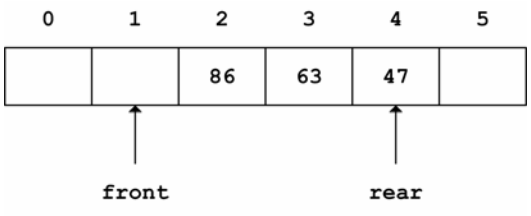
Δύο είναι οι κύριες λειτουργίες στην ουρά:

- **Εισαγωγή** (*enqueue*) στοιχείου στο πίσω άκρο της ουράς.
- **Εξαγωγή** (*dequeue*) στοιχείου από το εμπρός άκρο της ουράς.

Επομένως, για την υλοποίηση της ουράς χρειάζονται ένας πίνακας και δύο δείκτες, ο εμπρός (**front**) και ο πίσω (**rear**).

Για λόγους προγραμματιστικής διευκόλυνσης ο δείκτης **rear** δείχνει πάντα στο τελευταίο στοιχείο, ενώ ο δείκτης **front** δείχνει μία θέση πριν το πρώτο στοιχείο και, κατά συνέπεια, η ισότητα των δύο δεικτών αποδεικνύει ότι η ουρά είναι άδεια.

Εάν η ουρά υλοποιηθεί με πίνακα (στατική ουρά), απεικονίζεται στο **Σχήμα 10.2**. Θα πρέπει να τονιστεί ότι, κατ' αντιστοιχία με τη δομή της στοίβας, στην υλοποίηση της ουράς με πίνακα ο πίνακας αποτελεί μόνο το μέσο αποθήκευσης.



Σχήμα 10.3 *Απεικόνιση ουράς*

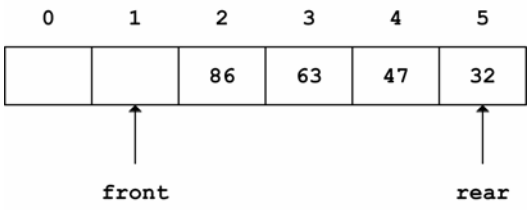
Όταν δημιουργείται η ουρά και πριν δεχτεί κάποιο δεδομένο, οι δύο δείκτες έχουν τιμή **-1**. Στο στιγμιότυπο της ουράς, που παρουσιάζεται στην **Εικόνα 10.3**, ο πίνακας διαθέτει **6** θέσεις (η πρώτη θέση έχει δείκτη θέσης **0**). Έχουν γίνει **5** εισαγωγές δεδομένων και **2** εξαγωγές.

Η υλοποίηση των δύο λειτουργιών δίνεται ακολούθως, όπου χάριν ευκολίας θεωρείται ότι τα δεδομένα είναι τύπου ακεραίου:

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

#define N 100        /* Το μέγεθος του πίνακα για την αποθήκευση της
                        ουράς*/

void push(int stack[], int *t, int obj);
int pop(int stack[], int *t);
/*-------------------------------------*/
int main()
{
  int q[N];
  int getStackValue;      /* Μεταβλητή υποδοχής των δεδομένων μετά
                             από pop */
  int front=-1,rear=-1;   /* Αρχικοποίηση των δεικτών, ώστε με την
                             έναρξη εισαγωγής και εξαγωγή δεδομένων
                             οι δείκτες να λάβουν τιμές συμβατές με
                             αριθμοδείκτες πίνακα */
  ............
  enqueue(q,&rear,<κάποιος ακέραιος>);
  ............
  enqueue(q,&rear,<κάποιος ακέραιος>);
  ............
  dequeue(q, int *front, int  rear);
  ............
  return 0;
}
/*-------------------------------------*/
void enqueue(int q[], int *r, int obj)
{
  if (*r==N-1)
  {
    printf( "ERROR! Queue is full..." );
    getchar();
  }
  else
```

```
    q[++(*r)]=obj;
}
/*-------------------------------------*/
void dequeue(int q[], int *f, int r)
{
  int x;
  if (*f==r)
    printf( "ERROR! Queue is empty...\n" );
  else
  {
    x=q[++(*f)];
    printf("%d has been deleted...",x);
  }
}
```

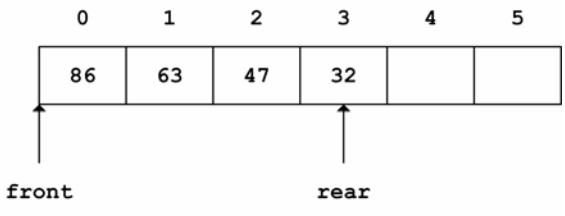
Η υλοποίηση της ουράς με πίνακα έχει ένα μειονέκτημα. Υπάρχει περίπτωση ο **rear** να φθάσει στο πάνω όριο του πίνακα, αλλά στην ουσία να μην υπάρχει υπερχείλιση (επειδή ο **front** θα έχει αυξηθεί), όπως φαίνεται στο **Σχήμα 10.4**.



Σχήμα 10.4 *Φαινομενική υπερχείλιση ουράς*

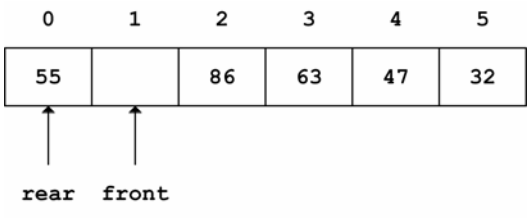
Στην πραγματικότητα, η υπερχείλιση είναι φαινομενική, καθώς στις αρχικές θέσεις του πίνακα, από τις οποίες έχουν γίνει εξαγωγές, υπάρχει ελεύθερος χώρος για την εισαγωγή νέων στοιχείων. Μία λύση θα ήταν να μεταφερθούν τα στοιχεία στο αριστερό άκρο του πίνακα (**Σχήμα 10.5**):

```
elements=front-rear;
first=front+1;
for (i=0;i<elements;i++)
  q[i]=q[first++];
front=-1;
rear=elements-1;
```



Σχήμα 10.5 *Άρση της φαινομενικής υπερχείλισης της ουράς με μετακίνηση στοιχείων στην αρχή της ουράς*

Μία πιο αποτελεσματική υλοποίηση θα ήταν η ουρά να αναδιπλώνεται, δηλαδή όταν ο **rear** = **N-1**, να επανατοποθετείται στο **0**. Η δομή αυτή ονομάζεται **κυκλική ουρά** *(circular queue)*. Στην ουρά του **Σχήματος 10.4** γίνεται εισαγωγή του δεδομένου **55**, το οποίο τοποθετείται πλέον στην αρχή του πίνακα που φιλοξενεί την ουρά, όπως φαίνεται στο **Σχήμα 10.6**.



Σχήμα 10.6 *Κυκλική ουρά*

10.3.1 Παράδειγμα

Το πρόγραμμα που ακολουθεί, αφορά στην εξυπηρέτηση αυτοκινήτων σε διόδια με χρήση ουράς. Συγκεκριμένα:

1. Η συνάρτηση `int display_menu()` υλοποιεί το παρακάτω μενού:

MENU	
======	
1.Car arrival	*(επιλογή για νέα άφιξη αυτοκινήτου)*
2.Car departure	*(επιλογή για αναχώριση αυτοκινήτου)*
3.Queue state	*(επιλογή για την εμφάνιση της τρέχουσας κατάστασης της ουράς)*
0.Exit	*(επιλογή για έξοδο)*

3. Η επιλογή θα επιστρέφεται στη `main()` και θα επιτελούνται οι ακόλουθες λειτουργίες:

(i) **Επιλογή 1:** Θα πληκτρολογούνται τα στοιχεία της πινακίδας του αυτοκινήτου σε αλφαριθμητικό `plate`, το οποίο θα τοποθετείται στο τέλος της ουράς, με χρήση της συνάρτησης

```
void enqueue(char **q, int *rear, char *obj)
```

(ii) **Επιλογή 2:** Το αυτοκίνητο που βρίσκεται πρώτο στην ουρά, θα διαγράφεται μαζί με ένα ανάλογο μήνυμα επιβεβαίωσης, με χρήση της συνάρτησης

```
void dequeue(char **q, int *front, int rear, int length)
```

όπου `length` το μήκος του αλφαριθμητικού που φιλοξενεί την πινακίδα (**7** αριθμοί και γράμματα + πιθανό κενό + μηδενικός χαρακτήρας = **9**)

(iii) **Επιλογή 3:** Θα εμφανίζονται, με τη σειρά, οι πινακίδες των αυτοκινήτων που παραμένουν στην ουρά, για να εξυπηρετηθούν, με χρήση της συνάρτησης

```
void printqueue(char **q, int front, int rear)
```

(iv) **Επιλογή 4:** Το πρόγραμμα θα τερματίζεται.

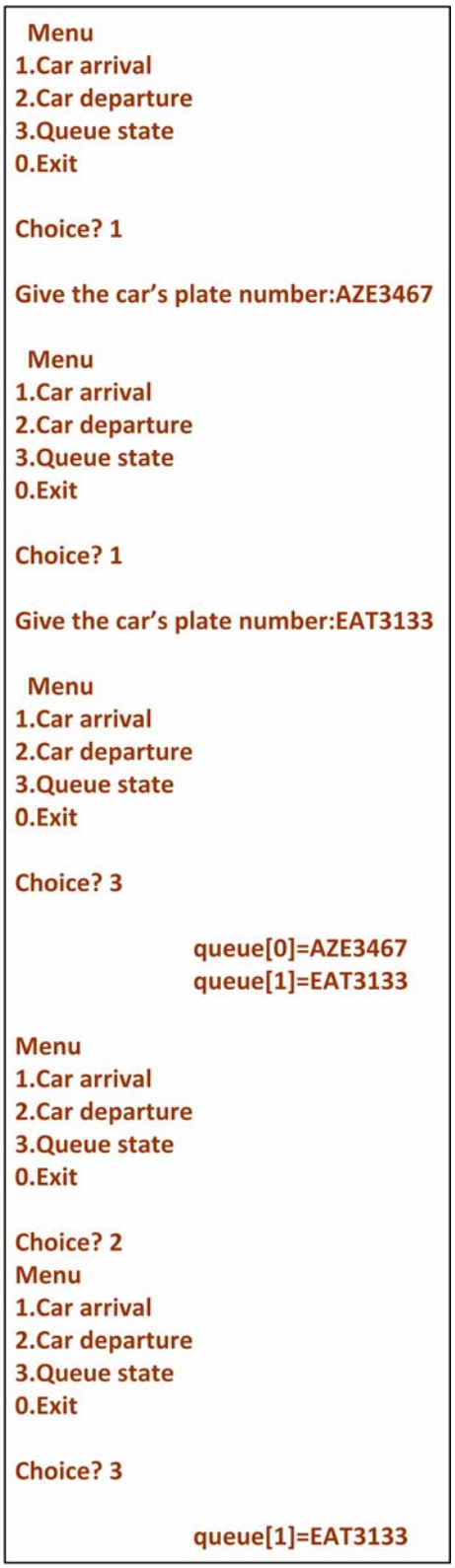
4. Οι πίνακες χαρακτήρων θα υλοποιηθούν με δυναμική δέσμευση μνήμης.

Εφόσον τα δεδομένα είναι αλφαριθμητικά, η ουρά θα αποθηκευτεί σε δισδιάστατο δυναμικό πίνακα χαρακτήρων και το δεδομένο «πινακίδα» σε μονοδιάστατο δυναμικό πίνακα. Το συνολικό πρόγραμμα είναι το ακόλουθο, ενώ η **Εικόνα 10.2** φιλοξενεί ένα στιγμιότυπο των αποτελεσμάτων:

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

#define PLATE_LENGTH 9
#define N 100

void enqueue(char **q, int *rear, char *obj);
```

```
void dequeue(char **q, int *front, int rear);
void printqueue(char **q, int front, int rear);
int display_menu();
/*-------------------------------------*/
int main()
{
  int i,front=-1,rear=-1,choice;
  char **q,*plate;

  plate=(char *)malloc(PLATE_LENGTH*sizeof(char));
  assert(plate!=NULL);
  q=(char **)malloc(N*sizeof(char *));
  assert(q!=NULL);
  for (i=0;i<N;i++)
  {
     q[i]=(char *)malloc(PLATE_LENGTH*sizeof(char));
     assert(q[i]!=NULL);
  }

  do
  {
     choice=display_menu();
     switch (choice)
     {
        case 1:
          printf( "\nGive the car's plate number:" );
          scanf( "%s",plate );
          enqueue(q,&rear,plate);
        break;
        case 2:
          dequeue(q,&front,rear);
        break;
        case 3:
          printqueue(q,front,rear);
        break;
        case 0:
          printf( "\nExiting ...\n" );
        break;
     }
  } while (choice!=0);

  for (i=(N-1);i>=0;i--)
     free(q[i]);
  free(q);
  free(plate);

  return 0;
}
/*-------------------------------------*/
int display_menu()
{
  int choice;
  do
  {
     printf( "\n\n  Menu:\n" );
```

```
            printf( "1.Car arrival\n" );
            printf( "2.Car departure\n" );
            printf( "3.Queue state\n" );
            printf( "0.Exit\n" );
            printf( "\nChoice? " );
            scanf( "%d",&choice );
          }  while  ((choice!=0)  &&  (choice!=1)  &&  (choice!=2)  &&
(choice!=3));
          return(choice);
        }
        /*-------------------------------------*/
        void enqueue(char **q, int *rear, char *obj)
        {
          if ((*rear)==(N-1))
          {
            printf( "ERROR! Queue is full..." );
            getchar();
          }
          else strcpy(q[++(*rear)],obj);
        }
        /*-------------------------------------*/
        void dequeue(char **q, int *front, int rear)
        {
          if ((*front)==rear)
            printf( "ERROR! Queue is empty...\n" );
          else
            printf( "%s has been served and deleted from the queue...",
q[++(*front)] );
        }
        /*-------------------------------------*/
        void printqueue(char **q, int front, int rear)
        {
          int i;
          for (i=(front+1);i<=rear;i++)
            printf( "\n\tqueue[%d]=%s",i,q[i] );
        }
```



Εικόνα 10.2 *Η έξοδος του προγράμματος του παραδείγματος 10.3.1*

10.4 Συνδεδεμένες λίστες

Οι στατικές δομές που μελετήθηκαν, παρουσιάζουν προβλήματα στην εισαγωγή και διαγραφή κόμβων, όταν αυτοί δε βρίσκονται στην αρχή ή το τέλος τους, καθώς και στην αποδοτική εκμετάλλευση της διαθέσιμης μνήμης. Κύριο χαρακτηριστικό των συνδεδεμένων λιστών (linked lists) είναι ότι αποτελούν ακολουθίες κόμβων, οι οποίοι πιθανότατα βρίσκονται σε απομακρυσμένες θέσεις μνήμης και η σύνδεσή τους γίνεται με δείκτες. Κατ' αυτόν τον τρόπο, η εισαγωγή και διαγραφή κόμβων σε οποιαδήποτε θέση της λίστας γίνεται πολύ πιο απλά. Ένα άλλο θετικό χαρακτηριστικό είναι ότι δεν απαιτείται εκ των προτέρων καθορισμός του μέγιστου αριθμού κόμβων της λίστας και η λίστα μπορεί να επεκταθεί ή να συρρικνωθεί κατά την εκτέλεση του προγράμματος (δυναμική δομή δεδομένων).

Οι κυριότερες κατηγορίες λιστών είναι:

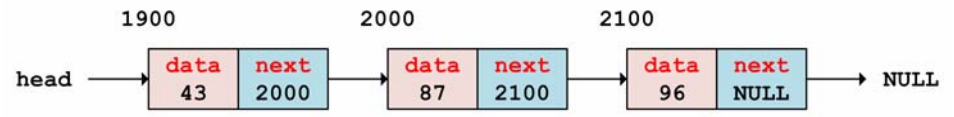
- Η απλά συνδεδεμένη λίστα (simply linked list).
- Η διπλά συνδεδεμένη λίστα (doubly linked list).
- Η κυκλική λίστα (circular list).

10.4.1 Απλά συνδεδεμένη λίστα

Η απλά συνδεδεμένη λίστα αποτελείται από μία αλυσίδα κόμβων, καθένας εκ των οποίων συνδέεται με τον επόμενο μέσω του τμήματος διεύθυνσης (next), δηλαδή το περιεχόμενο του τμήματος διεύθυνσης είναι η διεύθυνση του επόμενου κόμβου. Τοποθετείται ένας **δείκτης κεφαλής** (head) στον πρώτο κόμβο για να προσπελαύνεται η λίστα, ενώ ο δείκτης του τελευταίου κόμβου δείχνει στο **NULL**, για να εντοπίζεται το τέλος της λίστας. Ο κομβος υλοποιείται με την ακόλουθη δομή:

```
struct node
{
  <τύπος δεδομένου> <όνομα μεταβλητής>;
   struct node *next;
};
typedef struct node *PTR;
```

Ένα παράδειγμα απλά συνδεδεμένης λίστας με τρεις κόμβους που περιέχουν ακεραίους, δίνεται στο **Σχήμα 10.7**:



Σχήμα 10.7 *Απλά συνδεδεμένη λίστα τριών κόμβων*

Ο πρώτος κόμβος αποθηκεύεται στη διεύθυνση **1900**, στην οποία δείχνει ο **head**, και το μέλος **data** έχει περιεχόμενο το **43**. Ο πρώτος κόμβος συνδέεται με τον δεύτερο μέσω του μέλους **next**, το οποίο έχει ως περιεχόμενο τη διεύθυνση του δεύτερου κόμβου, **2000**. Με τον ίδιο τρόπο γίνεται η σύνδεση με τον τρίτο κόμβο, ο οποίος είναι τερματικός, καθώς το μέλος **next** έχει τιμή το **NULL**. Συνεπώς, η λίστα έρχεται σε επικοινωνία με το πρόγραμμα μόνο μέσω του δείκτη **head**.

Οι βασικές λειτουργίες της απλά συνδεδεμένης λίστας υλοποιούνται ακολούθως, όπου χάριν ευκολίας τα δεδομένα θεωρούνται ακέραιοι αριθμοί:

(α) **Δημιουργία λίστας με την προσθήκη ενός ή περισσότερων κόμβων**

```
PTR createList(PTR head)
{
  PTR current;
  int x;
  printf( "Give an integer, 0 to stop: " );
  scanf( "%i",&x );
```

```
  if (x==0)
     return NULL;
  else
  {
     /* Δημιουργία του πρώτου κόμβου*/
     head=malloc(sizeof(struct node));
     head->data=x;
     current=head;
     printf("Give an integer, 0 to stop:" );
     scanf( "%i",&x );
     while (x!=0) /* Εισαγωγή περισσότερων κόμβων. Κάθε νέος κόμβος
                     συνδέεται με το μέλος next του προηγούμενου */
     {
        current->next=malloc(sizeof(struct node));
        current=current->next;
        current->data=x;
        printf( "Give an integer, 0 to stop:" );
        scanf( "%i",&x );
     }
     current->next=NULL;  /* Σύνδεση του τερματικού κόμβου με το
                             NULL, είτε η λίστα αποτελείται από έναν
                             μόνο κόμβο είτε έχουν εισαχθεί
                             περισσότεροι. */
  }
  return head; /* Επιστροφή στην καλούσα συνάρτηση της διεύθυνσης
                  του πρώτου κόμβου της λίστας*/
}
```

(β) **Εισαγωγή στοιχείου σε διατεταγμένη λίστα (οι κόμβοι είναι τοποθετημένοι κατά αύξουσα τιμή των δεδομένων τους)**

```
PTR insertToList(PTR head, int x)
{
  PTR current,previous,newnode;
  int found;
  /* Δημιουργία νέου κόμβου */
  newnode=malloc(sizeof(struct node));
  newnode->data=x;
  newnode->next=NULL;
  /* Εάν η λίστα είναι άδεια, ο νέος κόμβος είναι η κεφαλή της. */
  if (head==NULL)
     head=newnode;
  /* Εάν η λίστα δεν είναι άδεια, αναζητείται το σημείο στο οποίο
     θα παρεμβληθεί ο νέος κόμβος, ώστε η λίστα να παραμείνει
     διατεταγμένη.  */
  else
  /* Περίπτωση τοποθέτησης του νέου κόμβου στην κεφαλή  */
     if (newnode->data < head->data)
     {
        newnode->next=head;
        head=newnode;
     }
  /* Περίπτωση τοποθέτησης του νέου κόμβου ανάμεσα σε δύο
     υφιστάμενους ή στο τέλος της λίστας */
     else
     {
```

```
            previous=head;
            current=head->next;
            found=0;
            while ((current!=NULL) && (found==0))
            {
               if (newnode->data < current->data)
                  found=1;
               else
               {
                  previous=current;
                  current=current->next;
               }
            }
            previous->next=newnode;
            newnode->next=current;
         }
      return head;
   }
```

(γ) **Διαγραφή στοιχείου**

```
PTR deleteFromList(PTR head, int x)
{
   PTR current,previous;
   int found;
   current=head;
   if (current==NULL)
   {
      printf( "Empty list...nothing to delete.\n" );
      getchar();
   }
   else
      if (x==head->data)   /* Διαγραφή του πρώτου κόμβου   */
      {
         /* Ο δεύτερος κόμβος γίνεται πλέον πρώτος */
         head=head->next;
         free(current);
      }
      else
      {
         previous=current;
         current=head->next;
         found=0;
         while ((current!=NULL) && (found==0))
         {
            if (x==current->data)
               found=1;
            else
            {
               previous=current;
               current=current->next;
            }
         }
         if (found==1)
         {
            previous->next=current->next;
```

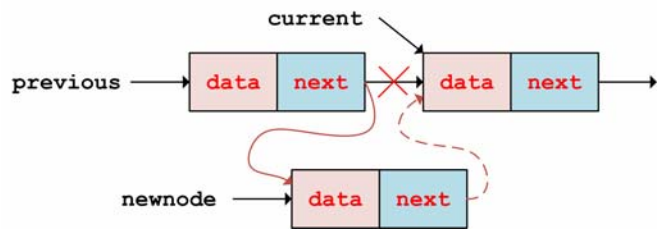
```
            free(current);
         }
         else
         {
            printf( "\nThe datum is not in the list." );
            getchar();
         }
      }
   return head;
}
```

(δ) **Εκτύπωση λίστας**

```
void printList(PTR head)
{
   PTR current;
   current=head;
   if (current==NULL)
      printf( "The list is empty.\n" );
   else
      while (current!=NULL)
      {
         printf( "%i ", current->data );
         current=current->next;
      }
}
```

Παρατηρήσεις:

1. Από τους ανωτέρω κώδικες προκύπτει ότι σημαντικό ρόλο στην εισαγωγή και διαγραφή κόμβου διαδραματίζει η σωστή σύνδεση, ώστε να μη χαθεί η συνέχεια της λίστας. Προς αυτήν την κατεύθυνση χρησιμοποιείται το ζεύγος δεικτών σε κόμβο `current` και `previous`, οι οποίοι κρατούν τις διευθύνσεις δύο διαδοχικών κόμβων και επιτρέπουν τη «γεφύρωσή» τους με τον νέο κόμβο κατά τη λειτουργία της εισαγωγής:



Σχήμα 10.8 *Εισαγωγή κόμβου*

Στο **Σχήμα 10.8** απεικονίζεται η διαδικασία εισαγωγής κόμβου ανάμεσα σε δύο υφιστάμενους. Με την πρόταση

```
previous->next=newnode;
```

συνδέεται ο προηγούμενος κόμβος με τον νέο κόμβο, διακόπτοντας τη σύνδεση με τον επόμενο κόμβο της λίστας, τη διεύθυνση του οποίου κρατά ο δείκτης `previous`. Με την πρόταση

```
newnode->next=current;
```

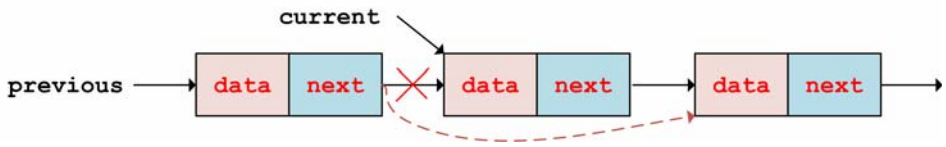
ο νέος κόμβος συνδέεται με τον επόμενο της λίστας, οπότε η σύνδεση διατηρείται.

Κατ' αντιστοιχία, όταν διαγράφεται ένας κόμβος, όπως απεικονίζεται στο **Σχήμα 10.9**, με την πρόταση

```
previous->next=current->next;
```

συνδέεται ο πρώτος με τον τρίτο κόμβο, ενώ ο δεύτερος δεν υπάγεται πλέον στη λίστα. Αυτός μέσω του δείκτη current διαγράφεται. Η μνήμη που καταλαμβάνει, απελευθερώνεται με την πρόταση

```
free(current);
```



Σχήμα 10.9 *Διαγραφή κόμβου*

2. Η διάσχιση της λίστας, π.χ. για την εκτύπωση των περιεχομένων της, γίνεται με χρήση του δείκτη σε κόμβο `current`, ο οποίος προωθείται από κόμβο σε κόμβο.

10.4.1.1 Παράδειγμα

Ο παρακάτω κώδικας δημιουργεί λίστα οκτώ κόμβων, η οποία θα τροφοδοτηθεί με τη σειρά δεδομένων **24, 31, 37, 1, 21, 25, 33, 30** (θεωρείται ότι η σειρά αριθμών βρίσκεται αποθηκευμένη σε προϋπάρχον αρχείο). Η συνάρτηση `PTR deleteEven(PTR head)` δέχεται τη διεύθυνση του πρώτου κόμβου της λίστας, διαγράφει τους κόμβους της λίστας που έχουν δεδομένα άρτιους αριθμούς και έχει επιστρεφόμενη τιμή τη διεύθυνση του πρώτου κόμβου της προκύψασας λίστας. Ακολούθως, εκτυπώνεται η νέα μορφή της λίστας.
Σημείωση: Ο αριθμός των κόμβων είναι δεδομένος (οκτώ), ωστόσο η συνάρτηση `PTR deleteEven()` είναι γενική, δηλαδή δεν θεωρούνται γνωστά ο αριθμός των υπαρχόντων κόμβων και τα περιεχόμενά τους.

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

#define N 8

struct node
{
  int data;
  struct node *next;
};
typedef struct node *PTR;

PTR deleteEven(PTR head);
/*-----------------------------------*/
int main()
{
  FILE *fin;
  PTR current,newnode,head;
  int i;
  fin=fopen( "data.dat","r" );
  printf( "\n\n\nINITIAL LIST\n" );
  /* Δημιουργία λίστας 8 κόμβων με περιεχόμενα από το αρχείο  */
  for (i=0;i<N;i++)
  {
    newnode=malloc(sizeof(struct node));
    assert(newnode!=NULL);
    fscanf( fin,"%d",&newnode->data );
```

```
            newnode->next=NULL;
            if (i==0)  /* Πρώτος κόμβος  */
               head=newnode;
            else /* Υπόλοιποι κόμβοι  */
               current->next=newnode;
            current=newnode;
            printf( "\nnode %2d:\taddress=%d\tdata=%d",i+1,current,current-
>data );
         }
         fclose(fin);
         printf( "\n\nDelete nodes with even data:\n" );
         head=deleteEven(head);
         /* Εκτύπωση της λίστας μετά τις διαγραφές */
         printf("\n\n\nFINAL LIST\n");
         i=1;
         current=head;
         while (current!=NULL)
         {
            printf(  "\nnode  %2d:\taddress=%d\tdata=%d",i,current,current-
>data );
            i++;
            current=current->next;
         }

         return 0;
      }
      /*-----------------------------------*/
      PTR deleteEven(PTR head)
      {
         int i;
         PTR current,previous;
         i=1;
         current=head;
         while (current!=NULL)
         {
            if ((current->data%2)==0) /* Συνθήκη διαγραφής */
            {
               printf( "\n\t\tNode no %d, with data value %d, will be de-
leted",i,current->data );
               if (current==head)   /* Διαγραφή του πρώτου κόμβου  */
               {
                  head=head->next;
                  free(current);
                  current=head;
               }
               else /* Διαγραφή κόμβου σε θέση διαφορετική της πρώτης */
               {
                  previous->next=current->next;
                  free(current);
                  current=previous->next;
               }
               previous=current;
            }
            else /* Μετάβαση στον επόμενο κόμβο για έλεγχο */
            {
```

```
            previous=current;
            current=current->next;
        }
        i++;
    }
    return(head);
}
```

```
INITIAL LIST

node 1:        address=7542792  data=24
node 2:        address=7542688  data=31
node 3:        address=7554256  data=37
node 4:        address=7554272  data=1
node 5:        address=7554288  data=21
node 6:        address=7554304  data=25
node 7:        address=7554320  data=33
node 8:        address=7554336  data=30

Delete nodes with even data:

            Node no 1, with data value 24, will be deleted
            Node no 1, with data value 30, will be deleted


FINAL LIST

node 1:        address=7542688  data=31
node 2:        address=7554256  data=37
node 3:        address=7554272  data=1
node 4:        address=7554288  data=21
node 5:        address=7554304  data=25
node 6:        address=7554320  data=33
```

Εικόνα 10.3 *Η έξοδος του προγράμματος του παραδείγματος 10.4.1.1*

Από τα αποτελέσματα συνάγεται ότι οι πρώτοι δύο κόμβοι τοποθετήθηκαν σε θέσεις μνήμης που απέχουν τόσο μεταξύ τους όσο και από τους υπόλοιπους έξι κόμβους, επιβεβαιώνοντας τη λογική κατανομής χώρου αποθήκευσης των δυναμικών δομών δεδομένων.

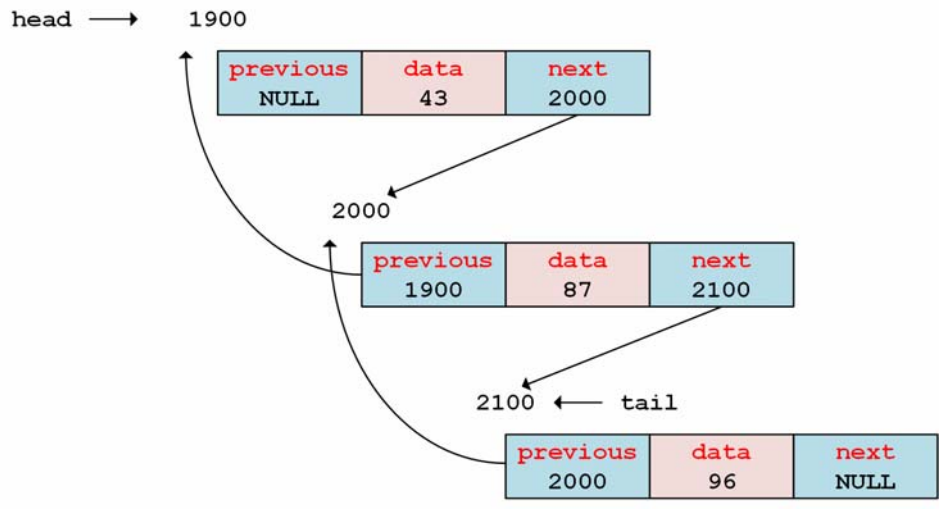
10.4.2 Διπλά συνδεδεμένη λίστα

Από την ανάλυση της απλά συνδεδεμένης λίστας διαπιστώνεται ότι, λόγω της χρήσης ενός μόνο δείκτη σε κάθε κόμβο, για να καταστεί προσβάσιμος ένας κόμβος της λίστας απαιτείται διάσχιση ολόκληρης της λίστας από την αρχή έως τον ζητούμενο κόμβο. Η διπλά συνδεδεμένη λίστα αποτελεί τροποποίηση της απλής, εισάγοντας ένα δεύτερο δείκτη σε κάθε κόμβο, ο οποίος δείχνει στον προηγούμενο κόμβο. Η διαχείριση της λίστας γίνεται πλέον με δύο δείκτες: τον **δείκτη κεφαλής** (head), ο οποίος δείχνει στον πρώτο κόμβο, και τον **δείκτη ουράς** (tail), ο οποίος δείχνει στον τελευταίο κόμβο. Ο κάθε κόμβος έχει ένα τμήμα δεδομένων και δύο τμήματα διεύθυνσης (**previous** και **next**). Ο κομβος υλοποιείται με την ακόλουθη δομή:

```
struct node
{
```

```
    <τύπος δεδομένου> <όνομα μεταβλητής>;
    struct node *next,*prev;
};
typedef struct node *PTR;
```

Ένα παράδειγμα διπλά συνδεδεμένης λίστας με τρεις κόμβους που περιέχουν ακεραίους, δίνεται στο **Σχήμα 10.10**:



Σχήμα 10.10 *Διπλά συνδεδεμένη λίστα τριών κόμβων*

Οι δύο βασικές λειτουργίες της εισαγωγής και της διαγραφής στοιχείου περιγράφονται ακολούθως:

(α) **Εισαγωγή στοιχείου**

Η εισαγωγή στοιχείου (τον κόμβο τον διαχειρίζεται ο **newnode**) στην αρχή της λίστας γίνεται ως εξής:

```
head->prev=newnode;
newnode->next=head;
head=newnode;
newnode->prev=NULL;
```

Αντίστοιχα γίνεται και η εισαγωγή κόμβου στο τέλος της λίστας:

```
tail->next=newnode;
newnode->prev=tail;
tail=newnode;
newnode->next=NULL;
```

Για την εισαγωγή κόμβου σε ενδιάμεση θέση θεωρείται ότι ο δείκτης **current** δείχνει στον κόμβο μετά τον οποίο θα γίνει η εισαγωγή, οπότε ο **current->next** δείχνει στον κόμβο που θα ακολουθεί τον εισαγόμενο:

```
newnode->next=current->next;
(current->next)->prev=newnode;
current->next=newnode;
newnode->prev=current->next;
```

(β) **Διαγραφή στοιχείου**

Η διαγραφή του κόμβου της κεφαλής γίνεται ως εξής:

```
current=head;          /* τοποθετείται σε δείκτη, ο οποίος
```

```
                                        θα επιτελέσει την απελευθέρωση μνήμης  */
head=head->next;
head->prev=NULL;
free(current);
```

Κατ' αντίστοιχο τρόπο διαγράφεται ο τελευταίος κόμβος της λίστας:

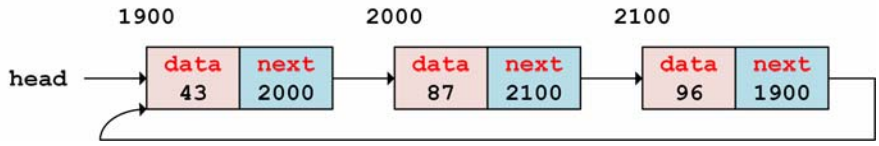
```
newnode=tail;
tail=tail->prev;
tail->next=NULL;
free(newnode);
```

Η διαγραφή κόμβου, ο οποίος βρίσκεται σε ενδιάμεση θέση, αποτελεί την Άσκηση 5 του κεφαλαίου.

Η διπλά συνδεδεμένη λίστα καθιστά ευκολότερη τη διαχείριση των κόμβων (εισαγωγή – διαγραφή), γεγονός ιδιαίτερα σημαντικό σε προβλήματα ταξινόμησης και αναζήτησης. Επίσης, τα περιεχόμενα των κόμβων προσπελαύνονται και από τις δύο κατευθύνσεις. Παρουσιάζει το μειονέκτημα της πολυπλοκότητας στη διαχείριση των κόμβων και της επιπρόσθετης απαιτούμενης μνήμης για κάθε κόμβο.

10.4.3 Κυκλική λίστα

Η κυκλική λίστα είναι μία απλά συνδεδεμένη λίστα, με τη διαφορά ότι το μέλος **next** του τελευταίου κόμβου δεν έχει τιμή **NULL** αλλά τη διεύθυνση του πρώτου κόμβου. Η δομή του κόμβου είναι ίδια με αυτή της απλά συνδεδεμένης λίστας. Ένα παράδειγμα κυκλικής λίστας με τρεις κόμβους που περιέχουν ακεραίους δίνεται στο **Σχήμα 10.11**:



Σχήμα 10.11 *Κυκλική λίστα τριών κόμβων*

Η διπλά συνδεδεμένη λίστα παρέχει ευελιξία λόγω του σχήματός της και βρίσκει εφαρμογή σε διαδικασίες, όπως ο διαμοιρασμός χρόνου από το λειτουργικό σύστημα. Ο διαμοιρασμός του χρόνου που οι εφαρμογές που τρέχουν σε έναν υπολογιστή και θα έχουν πρόσβαση στους πόρους του συστήματος, μπορεί να υλοποιηθεί με χρήση κυκλικής λίστας, καθώς δεν υπάρχουν δείκτες NULL. Έτσι, οι εφαρμογές θα διαμοιράζονται τον χρόνο με κυκλική εναλλαγή.

10.5 Εφαρμογές των συνδεδεμένων λιστών

Οι συνδεδεμένες λίστες εφαρμόζονται στην υλοποίηση των δομών δεδομένων της στοίβας και της ουράς, παρέχοντας τα πλεονεκτήματα της επεκτασιμότητας χωρίς πρακτικό περιορισμό υπερχείλισης και της εξοικονόμησης μνήμης, καθώς οι διαγραφές στοιχείων οδηγούν σε απελευθέρωση μνήμης.

10.5.1 Η στοίβα ως συνδεδεμένη λίστα

Η στοίβα υλοποιείται με έναν δείκτη **top**, ο οποίος αρχικοποιείται στο **NULL** και δείχνει ότι η στοίβα είναι άδεια. Ο δείκτης **top** διαδραματίζει τον ρόλο του δείκτη κεφαλής στην αρχή της λίστας. Όπως αναφέρθηκε ανωτέρω, η λειτουργία **push** δεν ελέγχει για υπερχείλιση, γιατί θεωρητικά η στοίβα μπορεί να έχει όσους κόμβους θέλουμε (αφού δημιουργείται δυναμικά). Επίσης, η λειτουργία **pop**, με τη βοήθεια της συνάρτησης `free()`, απελευθερώνει τον χώρο μνήμης που καταλαμβανόταν από τον κόμβο που διαγράφηκε.

Η εισαγωγή στοιχείου στην κορυφή της στοίβας υλοποιείται με την εισαγωγή νέου κόμβου στην αρχή της συνδεδεμένης λίστας. Αντίστοιχα, η διαγραφή στοιχείου από τη στοίβα υλοποιείται με τη διαγραφή του πρώτου κόμβου της συνδεδεμένης λίστας. Στο **Σχήμα 10.7** απεικονίζεται μία στοίβα ακέραιων αριθμών **3** θέσεων, με στοιχείο κορυφής το **43**.

(α) **Εισαγωγή στοιχείου στην κορυφή της στοίβας**

Η εισαγωγή στοιχείου (τον κόμβο τον διαχειρίζεται ο **newnode**) στην κορυφή της στοίβας γίνεται ως εξής:

```
PTR push(PTR top, int x)
{
  PTR newnode;
  int found;
  /* Δημιουργία νέου κόμβου */
  newnode=malloc(sizeof(struct node));
  newnode->data=x;
  /* Εάν η στοίβα είναι άδεια, ο νέος κόμβος είναι η κορυφή της.*/
  if (top==NULL)
  {
    top=newnode;
    newnode->next=NULL;
  }
  /* Εάν η στοίβα δεν είναι άδεια, τοποθετείται ο νέος κόμβος στην
κεφαλή */
  else
  {
    newnode->next=top;
    top=newnode;
  }
  return top;
}
```

(β) **Διαγραφή στοιχείου από τη στοίβα**

Ο κώδικας της διαγραφής στοιχείου από τη στοίβα είναι ο ακόλουθος:

```
PTR pop(PTR top, int *obj)
{
  PTR current;
  if (top==NULL)
  {
    printf( "Empty stack...nothing to delete.\n" );
    getchar();
  }
  else
  {
    current=top;
    top=top->next;
    *obj=current->data;
    free(current);
    return top;
  }
}
```

10.5.2 Η ουρά ως συνδεδεμένη λίστα

Η ουρά υλοποιείται με τη χρήση δύο δεικτών **front** και **rear**, οι οποίοι αρχικοποιούνται στην τιμή **NULL**. Η άδεια ουρά εκφράζεται με **front=NULL**. Όπως και στην περίπτωση της στοίβας, δεν χρειάζεται να γίνεται έλεγχος υπερχείλισης από τη στιγμή που η ουρά αυξάνεται δυναμικά.

(α) **Εισαγωγή στοιχείου στο τέλος της ουράς**

```
void enqueue(int obj, PTR *pf, PTR *pr)
{
  PTR newnode;
  newnode=malloc(sizeof(struct node));
  newnode->data=obj;
  newnode->next=NULL;
  /* Εάν η ουρά είναι άδεια, ο νέος κόμβος είναι η αρχή και το τέλος
     της. */
  if ((*pf)==NULL)
  {
     *pf=newnode;
     *pr=newnode;
  }
  /* Εάν η ουρά δεν είναι άδεια, τοποθετείται ο νέος κόμβος στο
     τέλος*/
  else
  {
     (*pr)->next=newnode;
     *pr=newnode;
  }
}
```

(α) **Διαγραφή στοιχείου από την αρχή της ουράς**

```
void dequeue(PTR *pf, PTR *pr)
{
  PTR current;
  if ((*pf)==NULL)
     printf( "\nEmpty queue. No elements to delete.\n" );
  else
  {
     current=*pf;
     *pf=(*pf)->next;
     if ((*pf)==NULL)
        *pr=*pf;
     printf( "\n%d has been deleted...\n",p->data );
     free(current);
  }
}
```

10.6 Παράδειγμα ανάπτυξης προγράμματος

Θα υλοποιηθεί μία εφαρμογή διαχείρισης πελατών τράπεζας, οι οποίοι περιμένουν στην ουρά, για να εξυπηρετηθούν από το ταμείο. Η τράπεζα διατηρεί ένα αρχείο με εγγραφές πελατών με τις ακόλουθες πληροφορίες:

- Αριθμός λογαριασμού.
- Ονοματεπώνυμο πελάτη.

- Ποσό λογαριασμού.

Το πρόγραμμα είναι καθοδηγούμενο από ένα μενού, με δύο βασικές επιλογές:

1. Άφιξη πελάτη στην τράπεζα.

2. Αναχώρηση πελάτη (εφόσον έχει εξυπηρετηθεί).

Κάθε φορά που προσέρχεται ένας πελάτης (επιλογή **1**), τοποθετείται στο τέλος μίας ουράς με τη λειτουργία **enqueue** (στην ουσία τοποθετείται ο αριθμός λογαριασμού του πελάτη, αν και θα μπορούσε να είναι κάποιος αριθμός προτεραιότητας).

Όταν ο χρήστης επιλέξει τη λειτουργία **2**, τότε αυτομάτως ο πελάτης που βρίσκεται στην αρχή της ουράς, προωθείται για να εξυπηρετηθεί, αφού πρώτα γίνει εξαγωγή του αριθμού λογαριασμού από την ουρά με τη λειτουργία **dequeue**. Βάσει του καταχωρημένου στην ουρά αριθμού λογαριασμού διεξάγεται αναζήτηση στο αρχείο των πελατών. Εάν βρεθεί η εγγραφή τότε, αφού εμφανιστούν τα περιεχόμενα της εγγραφής στην οθόνη, ο πελάτης ζητείται να επιλέξει μία εκ των δύο συναλλαγών:

1. Ανάληψη.

2. Κατάθεση.

Μετά την καταχώρηση του είδους της συναλλαγής ζητείται το ποσό της συναλλαγής, το οποίο και προστίθεται ή αφαιρείται από το τρέχον ποσό του λογαριασμού. Στο τέλος, η ενημερωμένη εγγραφή καταχωρείται πίσω στο αρχείο.

Εάν η αναζήτηση στο αρχείο είναι ανεπιτυχής, τότε ο αριθμός λογαριασμού αγνοείται και η ροή του προγράμματος επανέρχεται στο αρχικό μενού.

Τέλος, εάν ο χρήστης επιλέξει την έξοδο από το μενού, τότε το πρόγραμμα τερματίζει, αφού πρώτα εμφανίσει το πλήθος των πελατών που περιμένουν στην ουρά, για να εξυπηρετηθούν.

Με βάση την ανωτέρω περιγραφή της λειτουργίας της εφαρμογής, αρχικά ορίζεται το δεδομένο της εγγραφής πελάτη ως μία δομή:

```
struct accountT
{
   int numb;
   char name[15];
   float amount;
};
typedef struct account bankAccount;
```

Η αποθήκευση των εγγραφών των πελατών θα γίνει με χρήση ουράς. Πέραν της συνάρτησης **main()**, οι λειτουργίες του προγράμματος θα μεριστούν στις ακόλουθες συναρτήσεις:

- **`enqueue()`**: Εισάγει στοιχείο στο τέλος της ουράς.
- **`dequeue()`**: Εξάγει στοιχείο από την αρχή της ουράς.
- **`searchFile()`**: Αναζητά μία εγγραφή στο αρχείο βάσει του αριθμού λογαριασμού. Σε περίπτωση επιτυχίας, διαβάζει τα περιεχόμενα της εγγραφής.
- **`getSelection()`**: Επιλέγει το είδος της συναλλαγής (ανάληψη ή κατάθεση).
- **`updateAmount()`**: Ανάλογα με την επιλογή της συναλλαγής, ενημερώνει το ποσό που διαθέτει ο λογαριασμός.
- **`displayFile()`**: Εμφανίζει στην οθόνη τα περιεχόμενα του αρχείου.
- **`fillBlanks()`**: Διευθετεί την εμφάνιση των αποτελεσμάτων στην οθόνη, προσθέτοντας τα απαραίτητα κενά.

Το τελικό πρόγραμμα έχει την ακόλουθη μορφή:

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

```
#define N 100 /*Μέγεθος του πίνακα για την αποθήκευση της ουράς*/

struct accountT{
  int numb;
  char name[20];
  float amount;
};
typedef struct accountT bankAccount;
/*-----------------------------------*/
int getSelection(bankAccount account);
void updateAmount(bankAccount *p, int sel);
int searchFile(FILE *fp, int x, bankAccount *p);
void fillBlanks(char *s, int x);
void displayFile(FILE *fp);
void enqueue(int q[N], int *r, int obj);
int dequeue(int q[N], int *f, int rear);
/*-----------------------------------*/
int main()
{
   int i,choice,found,custCode,sel,reclen,ncust=0,endFlag=0;
   int q[N],front=-1,rear=-1;
   FILE *fp;
   bankAccount account;
   fp=fopen("data.dat","rb+");   assert(fp!=NULL);
   displayFile(fp);
   rewind(fp);
   do
   {
     printf( "\n\n  Menu\n" );
     printf( "1.Customer arrival\n" );
     printf( "2.Customer departure\n" );
     printf( "3.Exit\n" );
     printf( "\nChoice: " );
     scanf( "%d",&choice );
     switch (choice)
     {
        case 1:
           printf( "\nGive the account number:" );
           scanf( "%d",&custCode );
           enqueue(q,&rear,custCode);
       break;
       case 2:
           custCode=dequeue(q,&front,rear);
           if (custCode!=0)  {
              found=0;
              found=searchFile(fp,custCode,&account);
              if (found==1)   {
                 sel=getSelection(account);
                 updateAmount(&account,sel);
                 reclen=sizeof(bankAccount);
                 fseek(fp,-reclen,1);
                 fwrite(&account,sizeof(bankAccount),1,fp);
              }
              else   {
                printf( "\nERROR! Invalid account number!\n");
```

```
                getchar();
              }
              rewind(fp);
           }
           break;
           case 3:
              endFlag=1;
             break;
      }
    } while (endFlag==0);
    if (front==rear) printf( "No customers waiting in queue..." );
    else
    {
      for (i=front+1;i<=rear;i++)    ncust++;
      printf( "Number of customers waiting in queue: %d ",ncust );
    }
    getchar();
    fclose(fp);
    return 0;
}
/*-------------------------------------*/
void enqueue(int q[N], int *r, int obj)
{
    if ((*r)==N-1)
    {
      printf( "Queue is full..." );
      getchar();
    }
    else
      q[++(*r)]=obj;  }
/*-------------------------------------*/
int dequeue(int q[N], int *f, int rear)
{
    int temp;
    if ((*f)==rear)
    {
      printf( "Empty Queue...\n" );
      temp=0;
    }
    else
    {
      temp=q[++(*f)];
      printf( "Customer %d is served...",temp );
    }
    getchar();
    return temp;
}
/*-------------------------------------*/
int getSelection(bankAccount account)
{
    int sel;
    printf( "\nAccount no.: %i\n", account.numb );
    printf( "Name       : %s\n", account.name );
    printf( "Amount    : %6.2f\n", account.amount );
    printf( "\n1. Deposit\n" );
```

```
   printf( "2. Withdraw\n" );
   printf( "\n\nSelect transaction:" );
   do
   {
     scanf( "%d",&sel );
   } while ((sel<1) || (sel>2));
   return sel;
}
/*-------------------------------------*/
void updateAmount(bankAccount *p, int sel)
{
  float amt;
  printf( "Give the amount:" );
  scanf( "%f",&amt );
  if (sel==1)  p->amount+=amt;
  else
     if (amt<=p->amount)  p->amount-=amt;
     else
     {
       printf( "The amount is too large to withdraw!" );
       getchar();
     }
}
/*-------------------------------------*/
int searchFile(FILE *fp, int x, bankAccount *p)
{
   bankAccount account;
   int found=0;
   fread(&account,sizeof(bankAccount),1,fp);
   while ((!feof(fp)) && (found==0))
   {
     if (x==account.numb)    found=1;
     else    fread(&account,sizeof(bankAccount),1,fp);
   }
   *p=account;
   return found;
}
/*-------------------------------------*/
void fillBlanks(char *s, int x)
{
  int i;
  for (i=1;i<=(x-strlen(s));i++)
     strcat(s," ");
}
/*-------------------------------------*/
void displayFile(FILE *fp)
{
  bankAccount account;
  char temp[20]="NAME";
  fillBlanks(temp,20);
  printf( "\nThe contents of the file are:\n\n" );
  printf( "ACC.NO.     %s     AMOUNT\n",temp );
  fread(&account,sizeof(bankAccount),1,fp);
  while (!feof(fp))
  {
```

```
            fillBlanks(account.name,20);
            printf("%4d      %s      %7.2f\n",account.numb, account.name, ac-
count.amount);
            fread(&account,sizeof(bankAccount),1,fp);
          }
      }
```

```
The contents of the file are:

ACC.No.    NAME          AMOUNT
  101      Johnson         310.00
  102      Jameson        1310.00
  103      Taylor            0.00

  Menu
1.Customer arrival
2.Customer departure
3.Exit

Choice: 1

Give the account number:102

  Menu
1.Customer arrival
2.Customer departure
3.Exit

Choice: 2
Customer 102 is served...
Account no.      : 102
Name             : Jameson
Amount           : 1310.00

1.Deposit
2.Withdraw

Select transaction:1
Give the amount:150

  Menu
1.Customer arrival
2.Customer departure
3.Exit

Choice: 2
Empty Queue...
```

Εικόνα 10.4 *Αποτελέσματα του προγράμματος*

Στην **Εικόνα 10.4** παρουσιάζεται ένα στιγμιότυπο από την εκτέλεση του προγράμματος. Αρχικά, εμφανίζονται στην οθόνη οι εγγραφές που βρίσκονται στο αρχείο. Η πρώτη επιλογή είναι η άφιξη του πελάτη

με λογαριασμό **102** και η εισαγωγή του στην ουρά. Η επόμενη επιλογή είναι η αναχώρηση πελάτη, οπότε ο πελάτης **102** προχωρά στο ταμείο για συναλλαγή. Επιλέγεται η κατάθεση **150** ευρώ, οπότε εμφανίζονται τα στοιχεία του λογαριασμού του πελάτη (το ποσό είναι αυτό που υπάρχει πριν τη συναλλαγή). Επιλέγοντας εκ νέου αναχώρηση πελάτη, ενώ η ουρά έχει αδειάσει, εμφανίζεται σχετικό μήνυμα.

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Στις παρακάτω ερωτήσεις επιλέξτε μία από τις τέσσερις απαντήσεις.

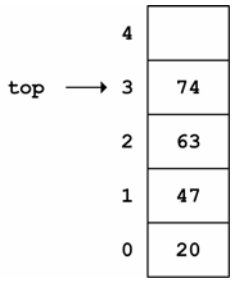
(1) Η λειτουργία pop(απώθηση) μίας στοίβας:
(α) ελέγχει αν η στοίβα είναι γεμάτη και τοποθετεί ένα νέο στοιχείο στην κορυφή της στοίβας.
(β) ελέγχει αν η στοίβα είναι άδεια και τοποθετεί ένα νέο στοιχείο στην κορυφή της στοίβας.
(γ) ελέγχει αν η στοίβα είναι γεμάτη και εξάγει το στοιχείο που βρίσκεται στην κορυφή της στοίβας.
(δ) ελέγχει αν η στοίβα είναι άδεια και εξάγει το στοιχείο που βρίσκεται στην κορυφή της στοίβας.

(2) Έστω το παρακάτω τμήμα προγράμματος, που δημιουργεί μία συνδεδεμένη λίστα.

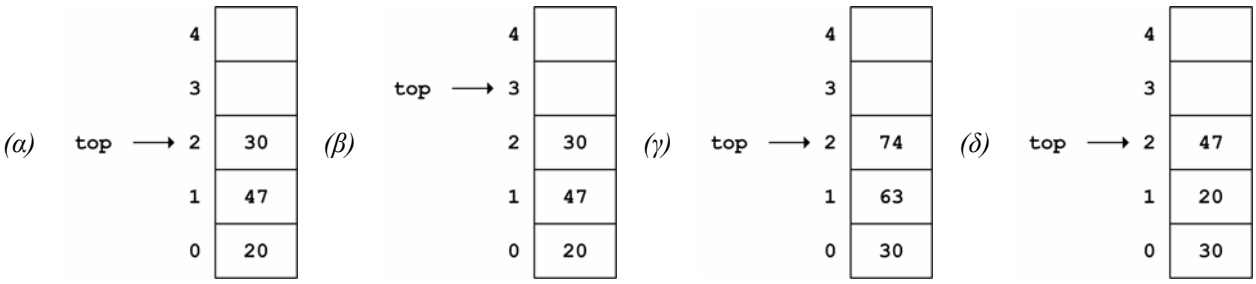
```
typedef struct node
{
   float data;
   struct node *next;
} *ptr;

ptr createList(ptr  p, float pin[]);

int main()
{
   float a[5]={9.1,7.2,3.8,3.1,6.9};
   ptr p=NULL;
   p=createList(p,a);
}

ptr createList(ptr  p, float pin[])
{
   int i;
   ptr current;
   for (i=0; i<5; i++)
   {
      current=malloc(sizeof(struct node));
      current->data=pin[i];
      current->next=p;
      p=current;
   }
   return p;
}
```

Με ποια σειρά αποθηκεύονται οι τιμές στους κόμβους, ξεκινώντας απο την αρχή της λίστας;
(α) 9.1, 7.2, 3.8, 3.1, 6.9
(β) 6.9, 3.1, 3.8, 7.2, 9.1
(γ) 3.1, 3.8, 6.9, 7.2, 9.1
(δ) 9.1, 7.2, 6.9, 3.8, 3.1

(3) Έστω η παρακάτω στοίβα:



Εκτελείται δύο φορές η λειτουργία της απώθησης (pop) και μία φορά η λειτουργία της ώθησης (push) του αριθμού **30**. Ποια είναι η νέα μορφή της στοίβας;



(4) Χρησιμοποιώντας τη δομή μίας στοίβας, υπολογίζεται η τιμή της παράστασης:

4 5 7 + 2 * +

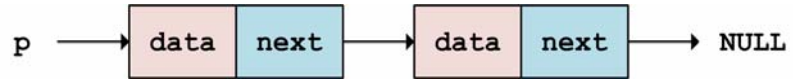
εφαρμόζοντας τους εξής κανόνες:

1. Αν το στοιχείο είναι τελεσταίος, τότε τοποθετείται σε μία στοίβα (push).
2. Αν το στοιχείο είναι τελεστής, τότε:
   - *i.* Εξάγονται διαδοχικά δύο τελεσταίοι από την κορυφή της στοίβας (pop).
   - *ii.* Εκτελείται η πράξη που δηλώνει ο τελεστής.
   - *iii.* Το αποτέλεσμα τοποθετείται στη στοίβα (push).

Ποια θα είναι η τιμή της παράστασης;

(α) 36
(β) 28
(γ) 24
(δ) 52

(5) Έστω η ακόλουθη συνδεδεμένη λίστα:



Θέλουμε να εισαχθεί ο παρακάτω νέος κόμβος ανάμεσα στους δύο κόμβους της λίστας.

Ποιος είναι ο κώδικας παρεμβολής του νέου κόμβου;

(α)
```
p=r;
r=p->next;
```
(α)
```
p->next=r;
r->next=p->next;
```
(α)
```
p->next=r->data;
r->next=p->next;
```
(α)
```
r->next=p->next;
p->next=r;
```

Ασκήσεις

Άσκηση 1

Να υλοποιηθεί η εφαρμογή της ενότητας 10.6 με χρήση δυναμικής ουράς.

Άσκηση 2

Να γραφεί κώδικας για τη δημιουργία λίστας έξι κόμβων, η οποία θα περιέχει ως δεδομένα τις δυνάμεις του 3 από το 2 έως το 7. Ακολούθως, να συγγραφεί αναδρομική συνάρτηση `void printListBackwards(PTR head)`, η οποία θα δέχεται τη διεύθυνση του πρώτου κόμβου της λίστας και θα εκτυπώνει τα περιεχόμενα της λίστας ανάποδα (από τον τελευταίο κόμβο προς τον πρώτο).

Άσκηση 3

Έστω μία υπάρχουσα και μη κενή συνδεδεμένη λίστα, η οποία περιέχει ως δεδομένα ακέραιους αριθμούς. Να γραφεί πρόγραμμα, το οποίο θα αναζητά τους κόμβους εκείνους που έχουν ως δεδομένο περιττό ακέραιο και όποτε βρει τέτοιον κόμβο, θα τον μεταφέρει από τη θέση, στην οποία βρίσκεται στην αρχή της λίστας (δηλαδή ο κόμβος αυτός θα γίνει σε εκείνο το στάδιο εκτέλεσης του προγράμματος ο πρώτος κόμβος της λίστας).

Άσκηση 4

Έστω μία υπάρχουσα και μη κενή συνδεδεμένη λίστα, η οποία περιέχει ως δεδομένα ακέραιους αριθμούς. Να γραφεί πρόγραμμα, το οποίο θα αναζητά τον κόμβο που έχει ως δεδομένο έναν αριθμό που διαβάζεται από το πληκτρολόγιο. Εάν βρεθεί τέτοιος κόμβος, θα διαγράφονται όλοι οι κόμβοι της λίστας από την αρχή της έως και αυτόν τον κόμβο.

Άσκηση 5

Να γραφεί πρόγραμμα για τη διαγραφή κόμβου από διπλά συνδεδεμένη λίστα, ο οποίος δεν βρίσκεται στην αρχή ή το τέλος της λίστας.

Άσκηση 6

Να γραφεί πρόγραμμα, το οποίο θα χρησιμοποιεί διπλά συνδεδεμένη λίστα για την αφαίρεση των σημείων στίξης από τα περιεχόμενα ενός υφιστάμενου αρχείου κειμένου.

Άσκηση 7

Να γραφεί πρόγραμμα, το οποίο θα διαχωρίζει μία υφιστάμενη απλά συνδεδεμένη λίστα με δεδομένα ακέραιους αριθμούς, σε δύο άλλες. Οι κόμβοι της αρχικής λίστας με δεδομένα θετικούς αριθμούς θα εντάσσονται στη μία εκ των δύο νέων λιστών, ενώ οι κόμβοι με στοιχεία αρνητικούς αριθμούς και το μηδέν θα εντάσσονται στην έτερη νέα λίστα.

Βιβλιογραφία κεφαλαίου

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

Κοίλιας, Χ. (2004), *Δομές Δεδομένων και Οργανώσεις Αρχείων*, Εκδόσεις Νέων Τεχνολογιών.

Μποζάνης, Π. (2006), *Δομές Δεδομένων*, Εκδόσεις Τζιόλα.

Παπουτσής, Ι. (2010), *Εισαγωγή στις Δομές Δεδομένων και στους Αλγόριθμους – Υλοποίηση σε C*, Εκδόσεις Σταμούλης.

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

Kalicharan, N. (2013), *Advanced Topics in C – Core Topics in Data Structures*, Apress.

Sahni, S. (2004), *Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++*, Εκδόσεις Τζιόλα.

Sedgewick, R. (2005), *Αλγόριθμοι σε C*, Εκδόσεις Κλειδάριθμος.

Wirth, N. (2004), *Αλγόριθμοι και Δομές Δεδομένων*, Εκδόσεις Κλειδάριθμος.

11. Διεπαφές

Σύνοψη
Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια της διασύνδεσης ή διεπαφής. Αρχικά, δίνονται η φιλοσοφία και τα γενικά χαρακτηριστικά των διεπαφών. Ακολούθως, παρουσιάζονται τα βήματα δημιουργίας μίας διεπαφής και τα κριτήρια για τον σχεδιασμό των διεπαφών. Τα θέματα της δημιουργίας διεπαφών και της διάσπασης του κώδικα σε πολλά αρχεία μελετώνται με μία σειρά τεσσάρων παραδειγμάτων, όπου γίνεται χρήση των εννοιών, των γλωσσικών κατασκευών και των εργαλείων που μελετήθηκαν στο παρόν και τα προηγούμενα κεφάλαια.

Λέξεις κλειδιά
διασύνδεση/διεπαφή, αρχείο κεφαλίδας, αρχείο βιβλιοθήκης, ifndef, απόκρυψη πληροφοριών.

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C – μεταβλητές – εκφράσεις – τελεστές – έλεγχος ροής προγράμματος – συναρτήσεις – πίνακες – δείκτες – δυναμική διαχείριση μνήμης – δομές δεδομένων – αρχεία

11.1 Η έννοια της διεπαφής

Στα προγράμματα που παρουσιάστηκαν στα προηγούμενα κεφάλαια, ο κώδικας φιλοξενείτο σε ένα μόνο αρχείο, π.χ. **`myProgram.c`**. Οι δηλώσεις συναρτήσεων και οι ορισμοί συναρτήσεων αποτελούσαν τμήματα του προγράμματος μέσα σε αυτόν τον ενιαίο χώρο. Από την άλλη πλευρά, ο κώδικας των πρότυπων συναρτήσεων βιβλιοθήκης (συναρτήσεις εισόδου–εξόδου, μαθηματικές συναρτήσεις κ.ο.κ.) δεν ήταν μεν ορατός από το πρόγραμμα, γινόταν όμως χρήση του μέσω των κλήσεων σε αυτές τις συναρτήσεις. Συνεπώς, έως τώρα στο ίδιο πρόγραμμα εμφανίζονταν δύο διαφορετικές προσεγγίσεις για την καταγραφή του κώδικα των συναρτήσεων: η άμεση προσέγγιση, όπου το σώμα της συνάρτησης βρισκόταν μέσα στο πρόγραμμα και η έμμεση – που αφορά στις συναρτήσεις βιβλιοθήκης – όπου δεν ενδιέφερε ο τρόπος υλοποίησης της συνάρτησης αλλά μόνο ο τρόπος χρήσης της και η κλήση της. Προφανώς, η δεύτερη προσέγγιση αφορούσε σε κώδικα που είχε γραφτεί σε διαφορετικό χρόνο και από διάφορους συγγραφείς, η χρησιμότητα του οποίου απαιτούσε την συμπερίληψή του χωρίς να υπάρχει ανάγκη ανάγνωσης και κατανόησής του.

Οι συναρτήσεις βιβλιοθήκης επικοινωνούν με ένα πρόγραμμα μέσω της οδηγίας προεπεξεργαστή **`#include`**, η οποία συνδέει στο πρόγραμμα αρχεία κεφαλίδας, όπως το αρχείο πρότυπης βιβλιοθήκης εισόδου–εξόδου **`stdio.h`**. Αυτά τα αρχεία αποτελούν ένα σύνορο μεταξύ δύο διακεκριμένων οντοτήτων: του αρχείου (βιβλιοθήκης), στο οποίο είναι γραμμένος ο κώδικας των συναρτήσεων, και του προγράμματος που χρησιμοποιεί τις συναρτήσεις. Επομένως, διασυνδέουν διαφορετικούς κώδικες, γι' αυτό και ονομάζονται **διασυνδέσεις** ή **διεπαφές** ή **διαπροσωπείες** (interfaces). Κάθε φορά που καλούνται συναρτήσεις αυτής της βιβλοθήκης, οι πληροφορίες περνούν διαμέσου αυτού του συνόρου. Η διεπαφή μεσολαβεί στην ανταλλαγή πληροφοριών μεταξύ της βιβλιοθήκης και του προγράμματος–χρήστη, το οποίο χρησιμοποιεί τις συναρτήσεις της.

Η ανάπτυξη και χρήση διεπαφών διευκολύνει τη συγγραφή κώδικα για τους ακόλουθους λόγους:

1. Επιτυγχάνεται επαναχρησιμοποίηση κώδικα, καθώς χρησιμοποιείται κώδικας που αναπτύχθηκε από τον ίδιο συγγραφέα ή από άλλους συγγραφείς, έχει αποσφαλματωθεί και είναι έτοιμος προς χρήση.

2. Ο προγραμματιστής δεν είναι υποχρεωμένος να έχει πρόσβαση στον κώδικα των συναρτήσεων που ορίζει μία διεπαφή. Απλώς χρειάζεται τις δηλώσεις των συναρτήσεων και επαρκή περιγραφή (υπό τη μορφή

σχολίων) του τις υλοποιούν, για να μπορεί να τις χρησιμοποιεί. Με αυτόν τον τρόπο αντιμετωπίζει τη συνάρτηση ως «μαύρο κουτί», όπου ο τρόπος υλοποίησης είναι αδιάφορος, και μπορεί να επικεντρωθεί στον πυρήνα του προγράμματός του. Η τακτική της απόκρυψης των εσωτερικών διεργασιών μίας βιβλιοθήκης ονομάζεται *απόκρυψη πληροφοριών* (information hiding).

3. Επιτυγχάνεται η οργάνωση και η ομαδοποίηση λειτουργιών σε συγκεκριμένα αρχεία, καθώς οι διεπαφές περιλαμβάνουν συναρτήσεις από συναφείς λειτουργίες. Για παράδειγμα, το αρχείο κεφαλίδας **`math.h`** αναφέρεται σε μαθηματικές συναρτήσεις και δεν περιλαμβάνει π.χ. συναρτήσεις διαχείρισης αλφαριθμητικών. Στις τελευταίες αναφέρεται το αρχείο κεφαλίδας **`string.h`**.

11.2 Δημιουργία διεπαφής

Η δημιουργία διεπαφής έγκειται στην υλοποίηση του αρχείου κεφαλίδας, π.χ. **`myLibray.h`**, και του αρχείου, στο οποίο θα βρίσκονται τα σώματα των συναρτήσεων, π.χ. **`myLibrary.c`**. Τα περιεχόμενα του αρχείου κεφαλίδας καθορίζονται από μία σειρά κανόνων:

1. Αρχικά παρατίθενται εκτενή σχόλια για τον σκοπό και τα περιεχόμενα της διεπαφής.

2. Μετά τα σχόλια περιέχονται οι γραμμές

```
#ifndef  _<όνομα διεπαφής>_h
#define  _<όνομα διεπαφής>_h
```

3. Η τελευταία γραμμή του αρχείου κεφαλίδας είναι

```
#endif
```

4. Ο κώδικας της διεπαφής θα περικλείεται ανάμεσα σε αυτές τις οδηγίες προεπεξεργαστή. Η υποθετική πρόταση που υλοποιούν ελέγχει κατά πόσον υπάρχει αρχείο κεφαλίδας με το ίδιο όνομα. Εάν δεν υπάρχει, τότε με την **`#define`** ορίζεται το όνομα και εκτελούνται οι επόμενες γραμμές του αρχείου κεφαλίδας. Πλέον, το συγκεκριμένο όνομα της διεπαφής έχει οριστεί και, εάν το ξανασυναντήσει ο μεταγλωττιστής, δεν θα χρειαστεί να ξαναδιαβάσει τη διεπαφή. Έτσι αποφεύγεται πολλαπλή ανάγνωση της ίδιας διεπαφής από τον μεταγλωττιστή. Η ανωτέρω τριάδα προτάσεων ονομάζεται **στερεότυπο** (boilerplate) και η παρουσία τους είναι υποχρεωτική σε κάθε διεπαφή.

5. Ο κώδικας της διασύνδεσης περιλαμβάνει τα **στοιχεία διεπαφής** (inteface entries). Αυτά είναι

(α) πρωτότυπα των συναρτήσεων (συνήθως συνοδευόμενα από σχόλια)

(β) (πιθανόν) σταθερές, π.χ. προσέγγιση του π ή η σταθερά του Planck,

(γ) (πιθανόν) τύποι δεδομένων οριζόμενοι από τον χρήστη, οι οποίοι χρησιμοποιούνται σε συναρτήσεις της βιβλιοθήκης.

Εάν οι συναρτήσεις αυτές χρειάζονται στοιχεία που βρίσκονται σε άλλες βιβλιοθήκες, π.χ. μία συνάρτηση χρειάζεται την **`fabs()`** που περιλαμβάνεται στο **`math.h`**, το σχετικό αρχείο κεφαλίδας πρέπει να συμπεριληφθεί μέσω της **`#include`**. Άπαξ και συμπεριληφθεί ένα αρχείο κεφαλίδας σε μία διεπαφή, δεν χρειάζεται να συμπεριληφθεί ξανά σε άλλο αρχείο που συναπαρτίζει το συνολικό πρόγραμμα.

Το αρχείο της βιβλιοθήκης των συναρτήσεων είναι συνώνυμο της διεπαφής και τη συμπεριλαμβάνει με την οδηγία προεπεξεργαστή **`#include`**, π.χ.

```
#include "myLibrary.h"
```

Σε αντιδιαστολή με τα πρότυπα αρχεία κεφαλίδας, όπου χρησιμοποιείται το ζεύγος **`< >`**, στις διασυνδέσεις που σχεδιάζουμε χρησιμοποιείται το ζεύγος **`" "`**. Η ίδια οδηγία προεπεξεργαστή εφαρμόζεται και στο κύριο πρόγραμμα, προκειμένου να συνδεθεί η βιβλιοθήκη με αυτό.

Η τελική μορφή μίας διεπαφής και του αρχείου βιβλιοθήκης που τη συνοδεύει είναι η ακόλουθη:

(i) **Αρχείο myLibrary.h**

```
/*  Σχόλια για τον σκοπό και τα περιεχόμενα της διεπαφής.  */
#ifndef  _myLibrary_h
#define  _ myLibrary _h
```

```
#include "mySecondLibrary.h"/* Άλλη βιβλιοθήκη, στοιχείο της οποίας
χρησιμοποιείται στην παρούσα βιβλιοθήκη  */
#include <πρότυπη βιλιοθήκη.h>   /* π.χ. string.h, στοιχείο της
οποίας χρησιμοποιείται στην παρούσα βιβλιοθήκη*/

/* Εάν απαιτείται: Δήλωση τύπων δεδομένων (π.χ. δομή)  */
#ifndef _dataTypeT_
#define _dataTypeT_
  typedef struct dataTypeT
  {
     δήλωση μεταβλητών-μελών
  };
#endif

/* Δήλωση συναρτήσεων */
<τύπος επιστρεφόμενης τιμής> firstFuncName(λίστα ορισμάτων);
<τύπος επιστρεφόμενης τιμής> secondFuncName (λίστα ορισμάτων);
......
<τύπος επιστρεφόμενης τιμής> lastFuncName(λίστα ορισμάτων);

#endif
```

(ii) **Αρχείο myLibrary.c**

```
#include "myLibrary.h"

/*   Σχόλια για τη λειτουργία της ακόλουθης συνάρτησης    */
<τύπος επιστρεφόμενης τιμής> firstFuncName(λίστα ορισμάτων);

/*   Σχόλια για τη λειτουργία της ακόλουθης συνάρτησης    */
<τύπος επιστρεφόμενης τιμής> secondFuncName (λίστα ορισμάτων);
......

/*   Σχόλια για τη λειτουργία της ακόλουθης συνάρτησης    */
<τύπος επιστρεφόμενης τιμής> lastFuncName(λίστα ορισμάτων);
```

Τα αρχεία βιβλιοθήκης που δημιουργούνται με τις διασυνδέσεις, πρέπει να μεταγλωττιστούν, όπως ακριβώς μεταγλωττίζεται το κύριο πρόγραμμα. Στα ολοκληρωμένα περιβάλλοντα ανάπτυξης προγράμματος προστίθενται στα *σχέδια προγράμματος* (Projects) και μεταγλωττίζονται όλα μαζί. Τα αρχεία κεφαλίδας δεν χρειάζονται μεταγλώττιση. Έτσι, παράγεται το αντικείμενο αρχείο (με κατάληξη **.obj** για συστήματα Windows και **.o** για συστήματα Unix) για κάθε πηγαίο αρχείο **.c**. Τα αντικείμενα αρχεία συνδέονται στον συνδέτη μαζί με τα πρότυπες βιβλιοθήκες και παράγεται το τελικό εκτελέσιμο αρχείο.

Θα πρέπει να προσεχθεί, ώστε να μην δημιουργείται κυκλική συμπερίληψη σε διασυνδέσεις. Δηλαδή, εάν μία διεπαφή **myLibrary.h** συμπεριλαμβάνει μία δεύτερη διεπαφή με την πρόταση **#include "mySecondLibrary.h"**, δεν είναι δυνατόν η **mySecondLibrary.h** να συμπεριλαμβάνει τη διεπαφή **myLibrary.h**.

11.3 Ιδιότητες διεπαφής

Όπως αναφέρθηκε προηγουμένως, οι διασυνδέσεις ελαττώνουν την πολυπλοκότητα του προγραμματισμού, κατ' αντιστοιχία με τις συναρτήσεις αλλά σε ανώτερο επίπεδο λεπτομέρειας: μία συνάρτηση παρέχει σε αυτόν που την καλεί την πρόσβαση σε ένα σύνολο βημάτων, τα οποία υλοποιούν μία συγκεκριμένη λειτουργία. Μία διεπαφή παρέχει στο κύριο πρόγραμμα (ή σε άλλα αρχεία βιλιοθήκης που τη χρησιμοποιούν)

πρόσβαση σε ένα σύνολο συναρτήσεων, οι οποίες υλοποιούν μία σειρά συναφών λειτουργιών. Ωστόσο, ο βαθμός απλοποίησης του προγραμματισμού σχετίζεται με το πόσο καλά σχεδιασμένη είναι η διεπαφή.

Για να σχεδιαστεί μία αποτελεσματική διεπαφή, πρέπει να ληφθούν υπόψη κάποια κριτήρια, τα οποία ορισμένες φορές έχουν μεταξύ τους ανταγωνιστική φύση. Οι διασυνδέσεις πρέπει εν γένει:

***(α)* Να έχουν ένα ενοποιητικό θέμα.** Οι συναρτήσεις που θα επιλεγούν να συμμετάσχουν σε μία διεπαφή, θα πρέπει να έχουν ένα συνεκτικό θέμα. Η βιβλιοθήκη **math** περιέχει αποκλειστικά μαθηματικές συναρτήσεις και κάθε συνάρτηση που εξάγεται από αυτή τη διεπαφή ταιριάζει με τον σκοπό της διεπαφής.

Επιπλέον, οι συναρτήσεις μίας διεπαφής θα πρέπει να συμπεριφέρονται με κατά το δυνατόν συνεπέστερο τρόπο. Για παράδειγμα, οι τριγωνομετρικές συναρτήσεις θα πρέπει να έχουν το ίδιο όρισμα σε ό,τι αφορά τη γωνία. Δεν πρέπει οι μισές να δέχονται μοίρες και οι μισές ακτίνια, έτσι ώστε ο προγραμματιστής που τις χρησιμοποιεί να γνωρίζει ότι η είσοδος είναι κοινή σε συναρτήσεις της ίδιας οικογένειας.

***(β)* Να είναι απλές**. Η απλότητα αποτελεί απαραίτητη προϋπόθεση για την ελάττωση της πολυπλοκότητας του προγραμματισμού, την οποία οι διασυνδέσεις στοχεύουν να προσφέρουν. Προς αυτή την κατεύθυνση κινείται η αρχή της απόκρυψης πληροφοριών, που αναφέρθηκε προηγουμένως. Ο προγραμματιστής–χρήστης πρέπει να δέχεται την κατάλληλη ποσότητα πληροφορίας. Ορισμένες φορές η αξία μίας διεπαφής δεν αναδεικνύεται από την πληροφορία που παρέχει, αλλά από αυτήν που αποκρύβει.

Η αρχή της απόκρυψης πληροφοριών έχει συνέπειες στη συγγραφή των διασυνδέσεων, καθώς δεν θα πρέπει να αποκαλύπτονται λεπτομέρειες της διεπαφής ούτε ακόμη και στα ερμηνευτικά σχόλια. Η διεπαφή έχει σκοπό να διευκολύνει τον προγραμματιστή που θα τη χρησιμοποιήσει και θα πρέπει να περιέχει μόνο όσα αυτός χρειάζεται να γνωρίζει.

Παρόμοια, οι συναρτήσεις μίας βιβλιοθήκης θα πρέπει να είναι σχεδιασμένες με απλό τρόπο. Εάν είναι δυνατόν να μειωθεί η λίστα των ορισμάτων ή να να απαλειφθούν περιπτώσεις που προκαλούν σύγχυση, θα είναι ευκολότερο για τον προγραμματιστή–χρήστη να κατανοήσει πώς θα χρησιμοποιήσει τις συναρτήσεις. Επιπρόσθετα, αποτελεί συνήθως καλή πρακτική να περιορίζεται ο συνολικός αριθμός των συναρτήσεων που περιλαμβάνονται στη διεπαφή, έτσι ώστε η τελευταία να μην καταντά χαώδης και να μην χάνεται ο προγραμματιστής–χρήστης μέσα σε πλήθος συναρτήσεων, μην μπορώντας να αποκτήσει αίσθηση του συνόλου.

(γ) **Να είναι επαρκείς**. Η διεπαφή θα πρέπει να παρέχει επαρκή λειτουγικότητα, ώστε να ικανοποιεί τις ανάγκες εκείνων που θα τη χρησιμοποιήσουν. Εάν μία λειτουργία απουσιάζει από μία διεπαφή, τότε – αργά ή γρήγορα – η τελευταία θα εγκαταληφθεί.

Είναι προφανές ότι η επάρκεια μπορεί να προσκρούει στην απλότητα, που αναφέρθηκε παραπάνω. Επομένως θα πρέπει να γίνει ένας συμβιβασμός ανάμεσα σε αυτά τα δύο χαρακτηριστικά, συνήθως ανάλογα με τα ιδιαίτερα χαρακτηριστικά της οικογένειας των συναρτήσεων που φιλοξενούνται στην εκάστοτε διεπαφή. Για παράδειγμα, εάν μία διεπαφή περιλαμβάνει συναρτήσεις για ένα σύστημα ελέγχου εναέριας κυκλοφορίας, η απλότητα έρχεται σε δεύτερη μοίρα σε σχέση με την ταχύτητα απόκρισης της συνάρτησης και η επάρκεια σε αυτήν την περίπτωση ταυτίζεται με την ταχύτητα στην επιστροφή σωστών απαντήσεων.

(δ) **Να είναι γενικές**. Μία καλοσχεδιασμένη διεπαφή θα πρέπει να είναι αρκετά ευέλικτη, ώστε να ικανοποιεί τις ανάγκες διαφορετικών κατηγοριών προγραμματιστών–χρηστών. Για να επιτευχθεί αυτό, η διεπαφή θα πρέπει να είναι αρκετά γενική, ώστε να λύνει ένα ευρύ φάσμα προβλημάτων.

(ε) **Να είναι σταθερές**. Οι συναρτήσεις που ορίζονται σε μία διεπαφή, θα πρέπει να εξακολουθούν να έχουν την ίδια ακριβώς μακροσκοπική εμφάνιση (είσοδοι – έξοδοι), ακόμη κι αν μεταβληθεί η υποκείμενη υλοποίηση. Εάν η μεταβολή της υλοποίησης οδηγεί σε αλλαγές στη συμπεριφορά μίας διεπαφής, οι προγραμματιστές–χρήστες θα πρέπει να αλλάξουν τα προγράμματά τους και αυτό θα θέσει σε κίνδυνο την αξία της διεπαφής. Η συνάρτηση **`sqrt(x)`**, που βρίσκεται στη διεπαφή **`math.h`**, υπολογίζει την τετραγωνική ρίζα του ορίσματός της, **`x`**. Η μαθηματική μέθοδος που υλοποιείται μέσα στη βιβλιοθήκη, είναι ένα ανάπτυγμα (π.χ. σειρά Taylor), το οποίο δεν είναι ορατό και – σε τελική ανάλυση – δεν ενδιαφέρει, εφόσον παρέχει το αποτέλεσμα με αποδεκτή ακρίβεια. Εάν μεταβληθεί ο τρόπος υπολογισμού, αλλά η συνάρτηση δατηρήσει τη μορφή της (ένα μόνο όρισμα και ίδιος τύπος επιστρεφόμενης τιμής), η διεπαφή παραμένει σταθερή. Εάν, όμως, η μεταβολή στον τρόπο υπολογισμού οδηγήσει σε μετονομασία της συνάρτησης ή μεταβολή του αριθμού ή/και του είδους των ορισμάτων της, τότε προκαλείται αναστάτωση, καθώς τα προγράμματα που τη χρησιμοποιούν πρέπει να τροποποιηθούν ανάλογα. Συμπερασματικά, οι

αλλαγές στις διασυνδέσεις θα πρέπει να επιχειρούνται σπάνια και μόνο με την ενεργό συμμετοχή του κοινού στο οποίο απευθύνονται.

Θα πρέπει να σημειωθεί ότι η προσθήκη μίας νέας συνάρτησης σε μία διεπαφή δεν διασαλεύει τη σταθερότητά της, καθώς πρόκειται για κάτι καινούριο, που δεν επηρεάζει τα υπάρχοντα προγράμματα που χρησιμοποιούν τη διεπαφή. Αυτή η διαδικασία ονομάζεται **επέκταση** (extending) της διεπαφής. Επομένως, εάν διαπιστωθεί ότι πρέπει να γίνουν εξελικτικές αλλαγές κατά τη διάρκεια της ζωής μίας διεπαφής, αυτές θα πρέπει να έχουν τη μορφή της επέκτασης. Υπό αυτή την έννοια, η δημιουργία μίας νέας συνάρτησης υπολογισμού της τετραγωνικής ρίζας, π.χ. `squareRoot(ορίσματα)`, με ταυτόχρονη διατήρηση της κλασσικής `sqrt(x)` είναι προτιμητέα από την αντικατάσταση της `sqrt(x)`.

11.4 Διάσπαση κώδικα σε πολλά αρχεία

Στην ενότητα αυτή θα μελετηθούν τα ζητήματα της δημιουργίας διασυνδέσεων και της διάσπασης του κώδικα σε πολλά αρχεία με τη βοήθεια παραδειγμάτων. Παραδείγματα που αναπτύχθηκαν στα προηγούμενα κεφάλαια, θα αποτελέσουν τη βάση για την παραγωγή δομημένου και επαναχρησιμοποιήσιμου κώδικα.

11.4.1 Παράδειγμα διαχείρισης αρχείων

Στην ενότητα 9.9 παρουσιάστηκε ένα πρόγραμμα διαχείρισης αρχείων, στο οποίο το έργο είχε κατανεμηθεί σε εννέα συναρτήσεις. Οι συναρτήσεις αυτές, οι οποίες είναι όλες συναφείς, μπορούν να ενταχθούν σε μία βιβλιοθήκη διαχείρισης αρχείων **fileHandling**. Η διεπαφή θα περιέχει τον ορισμό της δομής `RecordT`, καθώς αυτός είναι απαραίτητος στη λειτουργία των συναρτήσεων, τη σταθερά μέγιστο μήκος ονόματος `MAXLENGTH`, καθώς και μερικά εκ των αρχείων κεφαλίδας. Τα υπόλοιπα αρχεία κεφαλίδας θα περιλαμβάνονται στο κύριο πρόγραμμα. Ο συνολικός κώδικας περιλαμβάνει το κύριο πρόγραμμα **mainProgram.c**, το αρχείο κεφαλίδας **fileHandling.h** και το αρχείο βιβλιοθήκης **fileHandling.c**. Η διάρθρωσή του είναι η ακόλουθη:

(i) **mainProgram.c**

```
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#include "fileHandling.h"

const char *dirpath = "C:\\temp\\"; /* διαδρομή του καταλόγου στον
                                       οποίο βρίσκεται το αρχείο */
const char *file = "data.dat";      /* όνομα αρχείου */
/*-----------------------------------*/
int main(void)
{
  char filename[strlen(dirpath)+strlen(file)+1];
  strcpy(filename,dirpath);
  strcat(filename,file);
  /* Επιλογή λειτουργίας */
  char answer='q';
  while(true)
  {
    printf("\nChoose one of the following options:"
    "\nTo list the file contents enter L"
    "\nTo create a new file enter C"
    "\nTo add new records enter A"
    "\nTo update existing records enter U"
```

```
            "\nTo delete the file enter D"
            "\nTo end the program enter Q\n : ");
            scanf("\n%c", &answer);
            switch(tolower(answer))
            {
               case 'l':
                  listFile(filename);
               break;
               case 'c':
                  writeFile(filename,"wb+");
                  printf("\nFile creation complete.");
               break;
               case 'a':
                  writeFile(filename, "ab+");
                  printf("\nFile append complete.");
               break;
               case 'u':
                  updateFile(filename);
               break;
               case 'd':
                  printf("Are you sure you want to delete %s (y or n)? ",
filename);
                  scanf("\n%c", &answer);
                  if(tolower(answer)=='y')
                     remove(filename);
                     /* η συνάρτηση remove διαγράφει ένα αρχείο */
               break;
               case 'q': /* έξοδος από το πρόγραμμα */
                  printf("\nEnding the program.", filename);
                  return 0;
               default:
                  printf("Invalid selection. Try again.");
               break;
            }
         }

         return 0;
      }
```

(ii) **fileHandling.h**

```
      #ifndef _fileHandling_
      #define _fileHandling_

         #include <stdio.h>
         #include <assert.h>
         #include <stdbool.h>

         #define MAXLENGTH 80

         #ifndef _RecordT_
         #define _RecordT_
            struct RecordT
            {
               char name[MAXLENGTH];
```

```
        int age;
    };
#endif
/*-------------------------------------*/
void listFile(char *filename); /* Εμφανίζει τα περιεχόμενα του
αρχείου στην οθόνη  */

void updateFile(char *filename); /* Τροποποιεί τα περιεχόμενα του
αρχείου */

struct RecordT *getRecord(struct RecordT *precord);/* Λαμβάνει
δεδομένα (εγγραφές) από το πληκτρολόγιο  */

void getName(char *pname);  /* Διαβάζει  ένα  όνομα  από  το
πληκτρολόγιο */

void writeFile(char *filename, char *mode);        /* Έχει διπλή
λειτουργία: την εγγραφή σε ένα νέο αρχείο και την προσάρτηση σε
ένα υφιστάμενο αρχείο  */

void writeRecord(struct RecordT *precord, FILE *pFile); /* Γράφει
δεδομένα (εγγραφές) στο αρχείο */

struct RecordT *readRecord(struct RecordT *precord, FILE *pFile);
/* Λαμβάνει δεδομένα (εγγραφές) από το αρχείο  */

int findRecord(struct RecordT *precord, FILE *pFile); /* Αναζητά
μία εγγραφή στο αρχείο, βάσει μίας μεταβλητής-μέλος της εγγραφής*/

void duplicateFile(struct RecordT *pnewrecord, int index, char
*filename, FILE *pFile);  /* Αναπαράγει το αρχείο αντικαθιστώντας
μία  εγγραφή,  όταν  αυτή  έχει  διαφορετικό  μήκος  από  την
αντικαθιστώμενη */

#endif
```

***(iii)*fileHandling.c**

```
#include "fileHandling.h"

void listFile(char *filename)
{
......
}

void updateFile(char *filename)
{
......
}

struct RecordT *getRecord(struct RecordT *precord);
{
......
}

void getName(char *pname)
```

```
{
......
}

void writeFile(char *filename, char *mode)
{
......
}

void writeRecord(struct RecordT *precord, FILE *pFile)
{
......
}

struct RecordT *readRecord(struct RecordT *precord, FILE *pFile)
{
......
}

int findRecord(struct RecordT *precord, FILE *pFile)
{
......
}

void duplicateFile(struct RecordT *pnewrecord, int index, char
*filename, FILE *pFile)
{
......
}
```

Τα σώματα των συναρτήσεων, με σχετικό σχολιασμό παρατίθενται στην ενότητα 9.9.

Αποτίμηση της διεπαφής

Η διεπαφή είναι ενοποιημένη, καθώς όλες οι συναρτήσεις εντάσσονται στο ενιαίο θέμα της διαχείρισης στοιχείων του αρχείου ή ολόκληρου του αρχείου. Επιπρόσθετα, η διεπαφή μπορεί να θεωρηθεί ότι παρέχει κάποιο μέτρο απλούστευσης, καθώς αποκρύβει από τον προγραμματιστή–χρήστη ένα μεγάλο μέρος της πολυπλοκότητας των συναρτήσεων.

Σε ό,τι αφορά την επάρκεια και τη γενικότητά της, εφόσον οι συναρτήσεις σχετίζονται με τη δομή της εγγραφής ,η γενικότητα αναπόφευκτα περιορίζεται. Το πλήθος των λειτουργιών μπορεί να θεωρηθεί ικανοποιητικό για τη διαχείριση, επομένως η διεπαφή διαθέτει επάρκεια, χωρίς αυτό να σημαίνει ότι ίσως να απαιτείται κάποια επιπέον σχεδίαση, προκειμένου να ικανοποιηθούν πιο εξειδικευμένες λειτουργίες.

Τέλος, το ζήτημα της σταθερότητας δεν είναι ένα ερώτημα που αφορά τη φάση της σχεδίασης, αλλά μάλλον τον κύκλο της μακροπρόθεσμης συντήρησής της. Το σημαντικό ερώτημα είναι κατά πόσον η σχεδίαση προάγει με κάποιον τρόπο τη μακροπρόθεσμη σταθερότητα. Γενικά, μία διεπαφή που ικανοποιεί τα υπόλοιπα κριτήρια μπορεί, κατά πάσα πιθανότητα, να παραμείνει σταθερή, χρησιμοποιώντας στο μέλλον και την τεχνική της επέκτασης.

11.4.2 Παράδειγμα δυναμικής διαχείριση μνήμης

Στην ενότητα **7.7** και στα παραδείγματα **7.7.1** και **7.7.2** παρουσιάστηκαν συναρτήσεις για τη δυναμική δέσμευση και αποδέσμευση μνήμης. Αυτές οι συναρτήσεις, οι οποίες επιστρέφουν δείκτη στους προκαθορισμένους τύπους δεδομένων, μπορούν να ενσωματωθούν σε ένα αρχείο βιβλιοθήκης **allocateMemory**. Για λόγους οικονομίας του χώρου η δέσμευση θα αφορά σε δισδιάστατους δυναμικούς πίνακες, ωστόσο

μπορεί να επεκταθεί σε περισσότερες διαστάσεις. Ακολούθως, παρατίθενται το αρχείο κεφαλίδας **allocateMemory.h** και το αρχείο βιβλιοθήκης **allocateMemory.c**.

(i) **allocateMemory.h**

```
#ifndef _allocateMemory_
#define _allocateMemory_

  #include <assert.h>
  #include <stdlib.h>
  /*-----------------------------------*/
  char **allocChar_2(int lineNumber, int columnNumber); /* Δέσμευση
  μνήμης για δισδιάστατο πίνακα χαρακτήρων */

  int **allocInt_2(int lineNumber, int columnNumber);   /* Δέσμευση
  μνήμης για δισδιάστατο πίνακα ακεραίων */

  double **allocDouble_2(int lineNumber, int columnNumber);
  /*  Δέσμευση μνήμης για δισδιάστατο πίνακα αριθμών κινητής
  υποδιαστολής διπλής ακρίβειας  */

  float **allocFloat_2(int lineNumber, int columnNumber);
  /*  Δέσμευση μνήμης για δισδιάστατο πίνακα αριθμών κινητής
  υποδιαστολής απλής ακρίβειας*/

  void freeChar_2(char **parr, int lineNumber); /* Αποδέσμευση
  μνήμης για δισδιάστατο πίνακα χαρακτήρων */

  void freeInt_2(int **parr, int lineNumber); /* Αποδέσμευση  μνήμης
  για δισδιάστατο πίνακα ακεραίων  */

  void freeDouble_2(double **parr, int lineNumber);  /* Αποδέσμευση
  μνήμης για δισδιάστατο πίνακα αριθμών κινητής υποδιαστολής διπλής
  ακρίβειας */

  void freeFloat_2(float **parr, int lineNumber);  /* Αποδέσμευση
  μνήμης για δισδιάστατο πίνακα αριθμών κινητής υποδιαστολής απλής
  ακρίβειας */

#endif
```

(ii) **allocateMemory.c**

```
#include "allocateMemory.h"
/*-----------------------------------*/
char **allocChar_2(int lineNumber, int columnNumber)
{
  int i;
  char **parr;
  parr=(char **)malloc(lineNumber*sizeof(char *));
  assert(parr!=NULL);
  for (i=0;i<lineNumber;i++)
  {
    parr[i]=(char *)malloc(columnNumber*sizeof(char));
    assert(parr[i]!=NULL);
  }
```

```
  return(parr);
}
/*-----------------------------------*/
int **allocInt_2(int lineNumber, int columnNumber)
{
  int i,**parr;
  parr=(int **)malloc(lineNumber*sizeof(int *));
  assert(parr!=NULL);
  for (i=0;i<lineNumber;i++)
  {
    parr[i]=(int *)malloc(columnNumber*sizeof(int));
    assert(parr[i]!=NULL);
  }
  return(parr);
}
/*-----------------------------------*/
double **allocDouble_2(int lineNumber, int columnNumber)
{
  int i;
  double **parr;
  parr=(double **)malloc(lineNumber*sizeof(double *));
  assert(parr!=NULL);
  for (i=0;i<lineNumber;i++)
  {
    parr[i]=(double *)malloc(columnNumber*sizeof(double));
    assert(parr[i]!=NULL);
  }
  return(parr);
}
/*-----------------------------------*/
float **allocFloat_2(int lineNumber, int columnNumber)
{
  int i;
  float **parr;
  parr=(float **)malloc(lineNumber*sizeof(float *));
  assert(parr!=NULL);
  for (i=0;i<lineNumber;i++)
  {
    parr[i]=(float *)malloc(columnNumber*sizeof(float));
    assert(parr[i]!=NULL);
  }
  return(parr);
}
/*-----------------------------------*/
void freeChar_2(char **parr, int lineNumber)
{
  int i;
  for (i=(lineNumber-1);i>=0;i--)
     free(parr[i]);
  free(parr);
}
/*-----------------------------------*/
void freeInt_2(int **parr, int lineNumber)
{
  int i;
```

```
  for (i=(lineNumber-1);i>=0;i--)
     free(parr[i]);
  free(parr);
}
/*-------------------------------------*/
void freeDouble_2(double **parr, int lineNumber)
{
  int i;
  for (i=(lineNumber-1);i>=0;i--)
     free(parr[i]);
  free(parr);
}
/*-------------------------------------*/
void freeFloat_2(float **parr, int lineNumber)
{
  int i;
  for (i=(lineNumber-1);i>=0;i--)
     free(parr[i]);
  free(parr);
}
```

11.4.3 Παράδειγμα διαχείρισης απλά συνδεδεμένης λίστας

Στην υποενότητα **10.4.1** παρουσιάστηκαν συναρτήσεις για τη διαχείριση απλά συνδεδεμένης λίστας. Αυτές οι συναρτήσεις μπορούν να ενσωματωθούν σε ένα αρχείο βιβλιοθήκης **linkedListProcessing**. Η διεπαφή θα περιέχει τον ορισμό της δομής του κόμβου **node**, καθώς αυτός είναι απαραίτητος στη λειτουργία των συναρτήσεων, καθώς και μερικά εκ των αρχείων κεφαλίδας. Ακολούθως, παρατίθενται το αρχείο κεφαλίδας **linkedListProcessing.h** και το αρχείο βιβλιοθήκης **linkedListProcessing.c**.

(i) **linkedListProcessing.h**

```
#ifndef _linkedListProcessing_
#define _linkedListProcessing_

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

  #ifndef _node_
  #define _node_
     struct node
     {
        int data;
        struct node *next;
     };
     typedef struct node *PTR;
  #endif
  /*-------------------------------------*/
  PTR createList(PTR head); /* Δημιουργία λίστας με την προσθήκη
  ενός ή περισσότερων κόμβων  */

       PTR insertToList(PTR head, int x);   /* Εισαγωγή στοιχείου σε
   διατεταγμένη λίστα (οι κόμβοι είναι τοποθετημένοι κατά αύξουσα τιμή των
                              δεδομένων τους)  */

  PTR deleteFromList(PTR head, int x);   /* Διαγραφή στοιχείου   */
```

```
    void printList(PTR head);    /* Εκτύπωση των περιεχομένων της
    λίστας     */

#endif
```

(ii) **linkedListProcessing.c**

```
#include "linkedListProcessing.h"
/*-----------------------------------*/
PTR createList(PTR head)
{
......
}
PTR insertToList(PTR head, int x)
{
......
}
PTR deleteFromList(PTR head, int x)
{
......
}
void printList(PTR head)
{
......
}
```

Τα σώματα των συναρτήσεων, με σχετικό σχολιασμό παρατίθενται στην υποενότητα 10.4.1.

11.4.4 Παράδειγμα επεξεργασίας πινάκων

Στην ενότητα **7.8** αναπτύχθηκε πρόγραμμα επεξεργασίας τετραγωνικών πινάκων. Εάν οι συναρτήσεις ανάγνωσης (**getData()**) και εκτύπωσης (**`printData()`**) πίνακα θεωρηθεί ότι εμπίπτουν στις λειτουργίες διαχείρισης, τότε όλες οι συναρτήσεις της ενότητας **7.8** – με εξαίρεση τη συνάρτηση **`getSize()`** – ικανοποιούν το κριτήριο του ενοποιητικού θέματος. Εάν, όμως, επιχειρούσαμε να γράψουμε μία διεπαφή, η οποία θα επεξεργαζόταν μαθηματικά τον πίνακα, τότε μόνον οι συναρτήσεις υπολογισμού του ίχνους και αντιμετάθεσης γραμμών/στηλών θα εντάσσονταν σε μία τέτοια διεπαφή. Σε αυτήν θα είχαν θέση και άλλες λειτουργίες, όπως η αναστροφή και η αντιστροφή πίνακα, ο υπολογισμός της ορίζουσας και των ιδιοτιμών του κ.ά.

Η διεπαφή **sqMatrixProcessing** θα ενσωματώσει την πρότυπη βιβλιοθήκη μαθηματικών συναρτήσεων (αρχείο κεφαλίδας **math.h**) για τη χρήση της συνάρτησης **`fabs()`**. Τα υπόλοιπα αρχεία κεφαλίδας θα περιληφθούν στο κύριο πρόγραμμα, καθώς και στη διεπαφή **allocateMemory** (αναπτύχθηκε στην **11.4.2** για τη δυναμική δέσμευση και αποδέσμευση δισδιάστατου δυναμικού πίνακα), ένεκα του γεγονότος ότι το κύριο πρόγραμμα θα ζητά συναρτήσεις από την **allocateMemory**. Ο συνολικός κώδικας περιλαμβάνει το κύριο πρόγραμμα **mainProgram.c**, το αρχείο κεφαλίδας **sqMatrixProcessing.h** και το αρχείο βιβλιοθήκης **sqMatrixProcessing.c**. Η διάρθρωσή του είναι η ακόλουθη και τα αποτελέσματά του απεικονίζονται στην **Εικόνα 7.7**:

(i) **mainProgram.c**

```
#include <stdio.h>

#include "allocateMemory.h"
#include "sqMatrixProcessing.h"
```

```
int getSize(void);
/*-----------------------------------*/
int main()
{
  int i,j,col,size;
  float **pArr;

  size=getSize();
  pArr=allocFloat_2(size,size);

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

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

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

  freeFloat_2(pArr,size);

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

  return size;
}
```

(ii) **sqMatrixProcessing.h**

```
#ifndef _sqMatrixProcessing_
#define _sqMatrixProcessing_

  #include <math.h>
  /*-----------------------------------*/
  void getData(float **ptr, int size);   /* Ανάγνωση  των  στοιχείων
  του πίνακα*/

  void printData(float **ptr, int size); /* Εκτύπωση των πίνακα  */

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

```
  /* Εύρεση του στοιχείου της i γραμμής με τη μέγιστη απόλυτη τιμή
  και εμφάνισή της στην οθόνη */

  float trace(float **pArray, int size); /* Υπολογισμός του ίχνους
  του πίνακα*/

  void permuteColumns(float **pArray, int size, int column1, int
  column2); /* Αντιμετάθεση των στηλών column1 και column2 */


  void permuteLines(float **pArray, int size, int line1, int line2);
  /* Αντιμετάθεση των γραμμών line1 και line2 */

#endif
```

***(iii)*sqMatrixProcessing.c**

```
#include "sqMatrixProcessing.h"
/*-------------------------------------*/
void getData(float **ptr, int size)
{
  int i,j;
  for (i=0;i<size;i++)
     for (j=0;j<size;j++)
     {
        printf("\nA[%d][%d]:  ",i+1,j+1);
        scanf("%f",&ptr[i][j]);
     }
}
/*-------------------------------------*/
void printData(float **ptr, int size)
{
  int i,j;
  for (i=0;i<size;i++)
  {
     printf("\n");
     for (j=0;j<size;j++)  printf("\t%10.4f",ptr[i][j]);
  }
}
/*-------------------------------------*/
void getMax(float **pArray, int size, int i)
{
  int j,col;
  float maxim;
  maxim=fabs(pArray[i][0]);
  col=0;
  for (j=1;j<size;j++)
     if (fabs(pArray[i][j])>maxim)
     {
        maxim=fabs(pArray[i][j]);
        col=j;
     }
  printf( "\nLine %d: column %d, size=%f",i+1,col+1,fabs(pArray[i]
[col]) );
}
```

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

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

Ασκήσεις

Άσκηση 1

Με βάση τις υλοποιήσεις των συναρτήσεων διαχείρισης αλφαριθμητικών που δόθηκαν στις ενότητες 5.9 και 6.6, να γραφεί διεπαφή, που θα περιέχει τις συναρτήσεις αυτές.

Άσκηση 2

Με βάση τον κώδικα δέσμευσης και απελευθέρωσης μνήμης τρισδιάστατων δυναμικών πινάκων, που περιέχεται στο παράδειγμα 7.6.1, και τις συναρτήσεις δέσμευσης–αποδέσμευσης μνήμης του παραδείγματος 11.4.3, να *επεκταθεί* η διεπαφή allocateMemory (αρχεία allocateMemory.h και allocateMemory.c) ώστε να περιλαμβάνει τρισδιάστατους πίνακες.

Άσκηση 3

Λαμβάνοντας υπόψη:
***(α)* την υλοποίηση της ουράς με συνδεδεμένη λίστα στην υποενότητα 10.5.2,**
***(β)* την εκτύπωση των περιεχομένων συνδεδεμένης λίστας στην υποενότητα 10.4.1,**
***(γ)* τη διεπαφή allocateMemory της υποενότητας 11.4.3,**

να γραφεί πρόγραμμα για το πρόβλημα του παραδείγματος 10.3.1, στο οποίο θα δημιουργηθεί διεπαφή queueProcessing (αρχεία queueProcessing.h και queueProcessing.c) για τη φιλοξενία συναρτήσεων διαχείρισης ουράς.

Άσκηση 4
Να γραφεί πρόγραμμα για το πρόβλημα του παραδείγματος 5.3.1, στο οποίο οι πράξεις μεταξύ πινάκων θα υλοποιούνται με συναρτήσεις που θα υπάγονται σε διεπαφή matrixOper.

Άσκηση 5
Λαμβάνοντας υπόψη:
(α) **την υλοποίηση της στοίβας με συνδεδεμένη λίστα στην υποενότητα 10.5.1,**
(β) **την εκτύπωση των περιεχομένων συνδεδεμένης λίστας στην υποενότητα 10.4.1,**
(γ) **τη διεπαφή allocateMemory της υποενότητας 11.4.3,**
να γραφεί πρόγραμμα για το πρόβλημα του παραδείγματος 10.2.2, στο οποίο θα δημιουργηθεί διεπαφή stackProcessing (αρχεία stackProcessing.h και stackProcessing.c) για τη φιλοξενία αποκλειστικά των συναρτήσεων διαχείρισης ουράς.

Άσκηση 6
Να γραφεί πρόγραμμα, το οποίο θα επιτελεί τα ακόλουθα:
(α) **Θα δέχεται από το πληκτρολόγιο έναν ακέραιο αριθμό `n`, οποίος εκφράζει τη διάσταση διανυσμάτων. Ακολούθως, θα δεσμεύεται δυναμικά μνήμη για τρία διανύσματα ακεραίων `a`, `b`, `c`. Θα γίνεται έλεγχος, ώστε η διάσταση που θα δώσει ο χρήστης να είναι μεγαλύτερη ή ίση του 3 και μικρότερη ή ίση του 6.**
(β) **Θα καλείται η συνάρτηση `void readVector(int *vector, int n)`, η οποία θα λαμβάνει από το πληκτρολόγιο τιμές για τα στοιχεία μονοδιάστατου πίνακα `n` θεσεων, έτσι ώστε έμμεσα να αποκτήσουν τιμές τα διανύσματα `a` και `b`.**
(γ) **Θα καλείται η συνάρτηση `void add_vectors(int *vector1, int *vector2, int *vector3, int n)`, η οποία θα αθροίζει τα περιεχόμενα των διανυσμάτων `a` και `b` και θα τα αποθηκεύει στο διάνυσμα `c`.**
(δ) **Θα καλείται η συνάρτηση `int innerProduct(int *vector1, int *vector2, int n)`, η οποία θα υπολογίζει το εσωτερικό γινόμενο των διανυσμάτων `a` και `b` και θα επιστρέφει το αποτέλεσμα στη συνάρτηση *main()*.**
(ε) **Τα περιεχόμενα των διανυσμάτων `a`, `b`, `c` και το εσωτερικό γινόμενο των δύο πρώτων θα εμφανίζονται στην οθόνη.**
(στ) **Οι συναρτήσεις θα βρίσκονται μέσα στη διεπαφή vectorFuncs.**

Άσκηση 7
Να *επεκταθεί* η διεπαφή sqMatrixProcessing (αρχεία sqMatrixProcessing.h και sqMatrixProcessing.c) με την προσθήκη των ακόλουθων λειτουργιών:
(α) **Εξαγωγή του ανάστροφου ενός τετραγωνικού πίνακα (κάθε στοιχείο A_{ij} ενός πίνακα Α γίνεται το στοιχείο A_{ji} του ανάστροφου του πίνακα Α).**
(β) **Έλεγχος κατά πόσον ένας τετραγωνικός πίνακας Α είναι συμμετρικός (κάθε στοιχείο A_{ij} ενός πίνακα Α ισούνται με το στοιχείο A_{ji} του πίνακα).**

Βιβλιογραφία κεφαλαίου

Collopy, D. (2002), *Introduction to C Programming: A Modular Approach*, Prentice Hall.

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Hanly, J. & Koffman, E. (2013), *Problem Solving and Program Design in C*, 7th ed., Pearson.

Hartel, P. & Muller, H. (1997), *Functional C*, Addison-Wesley.

King, K. (2008), *C Programming: A Modern Approach*, 2nd ed., W.W. Norton & Company.

Kochan, S. (2005), *Programming in C*, 3rd ed., SAMS Publishing.

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

Βιβλιογραφία

Ελληνόγλωσση

Θραμπουλίδης, Κ. (2000), *Γλώσσες Προγραμματισμού (Τόμος Δ)*, ΕΑΠ.

Θραμπουλίδης, Κ. (2002), *Διαδικαστικός Προγραμματισμός - C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα.

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

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

Κοίλιας, Χ. (2004), *Δομές Δεδομένων και Οργανώσεις Αρχείων*, Εκδόσεις Νέων Τεχνολογιών.

Μποζάνης, Π. (2006), *Δομές Δεδομένων*, Εκδόσεις Τζιόλα.

Παπουτσής, Ι. (2010), *Εισαγωγή στις Δομές Δεδομένων και στους Αλγόριθμους – Υλοποίηση σε C*, Εκδόσεις Σταμούλης.

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

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

Deitel, H. & Deitel, P. (2005), *Ασκήσεις - Προγράμματα σε C*, Εκδόσεις Γκιούρδα.

Deitel, H. & Deitel, P. (2014), *C Προγραμματισμός*, 7η έκδοση, Εκδόσεις Γκιούρδα.

Kernighan, B. & Ritchie, D. (1990), *Η γλώσσα προγραμματισμού C*, Εκδόσεις Κλειδάριθμος.

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

Sahni, S. (2004), *Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++*, Εκδόσεις Τζιόλα.

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

Sedgewick, R. (2005), *Αλγόριθμοι σε C*, Εκδόσεις Κλειδάριθμος.

Wirth, N. (2004), *Αλγόριθμοι και Δομές Δεδομένων*, Εκδόσεις Κλειδάριθμος.

Ξενόγλωσση

Collopy, D. (2002), *Introduction to C Programming: A Modular Approach*, Prentice Hall.

Gabrielli, M. & Martini, S. (2010), *Programming Languages: Principles and Paradigms*, Springer.

Hanly, J. & Koffman, E. (2013), *Problem Solving and Program Design in C*, 7th ed., Pearson.

Hartel, P., Muller, H. (1997), *Functional C*, Addison-Wesley.

Horton, I. (2006), *Beginning C – from Novice to Professional*, 4th ed., Apress.

Kalicharan, N. (2013), *Advanced Topics in C – Core Topics in Data Structures*, Apress.

Kelley, A. & Pohl, I. (1998), *A Book on C*, 4th ed , Addison-Wesley.

Kernighan, B. & Pike, R. (1999), *The Practice of Programming*, Addison-Wesley.

King, K. (2008), *C Programming: A Modern Approach*, 2nd ed., W.W. Norton & Company.

Kochan, S. (2005), *Programming in C*, 3rd ed., SAMS Publishing.

Lohr, S. (2001), *Go To*, Basic Books.

Prata, S. (2014), *C Primer Plus*, 6th ed., Addison-Wesley.

Prinz, P. & Crawford, T. (2005), *C in a Nutshell*, O'Reilly.

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

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

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

Ευρετήριο