ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΣΗΜΕΙΩΣΕΙΣ
ΕΙΣΑΓΩΓΗΣ ΣΤΟΝ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ

ΠΑΝΑΓΙΩΤΗΣ ΣΤΑΜΑΤΟΠΟΥΛΟΣ

ΑΘΗΝΑ – 2022

Περιεχόμενο του μαθήματος

- Γενικά περί υπολογιστών και προγραμματισμού υπολογιστών
  - Ιστορική αναδρομή
  - Η δομή του υπολογιστή
  - Η πληροφορία στον υπολογιστή
- Λογισμικό και γλώσσες προγραμματισμού
  - Απαιτήσεις από μία διαδικαστική γλώσσα προγραμματισμού
  - Εκτελέσιμα προγράμματα
  - Μεταγλώττιση και σύνδεση
- Η γλώσσα προγραμματισμού C
  - Προγραμματιστικά περιβάλλοντα για την C
  - Ο μεταγλωττιστής gcc
  - Παραδείγματα απλών προγραμμάτων στην C
  - Χαρακτηριστικά και δυνατότητες της C
    - Μεταβλητές, σταθερές, τύποι και δηλώσεις
    - Εντολές αντικατάστασης, τελεστές και παραστάσεις
    - Η ροή του ελέγχου
    - Δομή προγράμματος, συναρτήσεις και εξωτερικές μεταβλητές
    - Εμβέλεια και χρόνος ζωής μεταβλητών

    - Αναδρομή
    - Διευθύνσεις θέσεων μνήμης, δείκτες και πίνακες
    - Δυναμική δέσμευση μνήμης
    - Συμβολοσειρές
    - Πίνακες δεικτών, δείκτες σε δείκτες και πολυδιάστατοι πίνακες
    - Δείκτες σε συναρτήσεις
    - Ορίσματα γραμμής εντολών
    - Απαριθμήσεις, δομές, αυτο-αναφορικές δομές (λίστες, δυαδικά δέντρα), ενώσεις, πεδία bit και δημιουργία νέων ονομάτων τύπων
    - Είσοδος και έξοδος
    - Χειρισμός αρχείων
    - Προεπεξεργαστής της C και μακροεντολές
  - Αλγόριθμοι ταξινόμησης πινάκων και αναζήτησης σε πίνακες
  - Οδηγίες σωστού προγραμματισμού
  - Συχνά προγραμματιστικά λάθη στην C
- Εργαστηριακές ασκήσεις και εργασίες για κατ΄ οίκον εκπόνηση

Εισαγωγικά θέματα

- Γενικά περί υπολογιστών
- Γενικά περί προγραμματισμού υπολογιστών
- Ιστορική αναδρομή
- Κάποια υπολογιστικά προβλήματα
- Πώς να μάθουμε να προγραμματίζουμε;
- Η δομή του υπολογιστή
- Η πληροφορία στον υπολογιστή
- Λογισμικό και γλώσσες προγραμματισμού

Γενικά περί υπολογιστών

- Υπολογιστής είναι ένα τεχνητό κατασκεύασμα που έχει την ικανότητα να επεξεργάζεται ένα σύνολο από *δεδομένα* που του δίνονται και να παράγει τα απαιτούμενα *αποτελέσματα*.
- Το είδος της επεξεργασίας που πρέπει να γίνει επάνω στα δεδομένα προσδιορίζεται από ένα *πρόγραμμα* με το οποίο έχουμε τροφοδοτήσει τον υπολογιστή και το οποίο αποτελείται από στοιχειώδεις εκτελέσιμες *εντολές* που είναι σε θέση να φέρει σε πέρας ο υπολογιστής.
- Γιατί χρειαζόμαστε υπολογιστές;
  - Για τη γρήγορη εκτέλεση χρονοβόρων αριθμητικών υπολογισμών, για παράδειγμα πόσο είναι το π, αν $\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \frac{1}{6^2} + \frac{1}{7^2} + \frac{1}{8^2} + \ldots$;
  - Για την αποδοτική διαχείριση ογκωδών δεδομένων, για παράδειγμα τραπεζικών λογαριασμών.
  - Και για τα δύο προηγούμενα, ταυτόχρονα, για παράδειγμα στην πρόγνωση καιρού.
- Η συγγραφή προγραμμάτων για υπολογιστές αναφέρεται με τον γενικό όρο *προγραμματισμός υπολογιστών*.

Γενικά περί προγραμματισμού υπολογιστών

- Στη γενική περίπτωση, προγραμματισμός (ενός υπολογιστή) είναι ο σαφής καθορισμός μίας διαδικασίας, σαν ένα σύνολο από εντολές (το πρόγραμμα), που περιγράφει λεπτομερώς τα βήματα που πρέπει να γίνουν για να επιλυθεί ένα πρόβλημα υπολογισμού.
- Τα προγράμματα για ένα υπολογιστή αποτελούν το *λογισμικό* του (software), ενώ η συσκευή του υπολογιστή είναι το *υλικό* (hardware).
- Ένα πρόγραμμα τροφοδοτείται σ΄ ένα υπολογιστή, αυτός το εκτελεί και παράγει το αποτέλεσμα της επίλυσης του προβλήματος.
- Η συγγραφή ενός προγράμματος για την επίλυση ενός προβλήματος προϋποθέτει την επινόηση ενός *αλγορίθμου*, δηλαδή μίας σαφώς καθορισμένης διαδικασίας μέσω ενός συνόλου εκτελέσιμων βημάτων, που είναι εγγυημένο ότι μετά από ένα πεπερασμένο πλήθος βημάτων που θα εκτελεσθούν θα τερματίσει.
- Ένα πρόγραμμα είναι η διατύπωση ενός αλγορίθμου σε μία συγκεκριμένη *γλώσσα προγραμματισμού*.

Ιστορική αναδρομή

- Άβακας, συσκευή προσθαφαιρέσεων με μπίλιες (1000 π.Χ.)
- Ο Μηχανισμός των Αντικυθήρων (75 π.Χ.)
- Muhammad ibn Musa al-Khwarizmi, πρότεινε την έννοια της διαδικασίας για αυτόματους υπολογισμούς (800 μ.Χ.)
- Μηχανικοί υπολογιστές για αριθμητικούς υπολογισμούς (Blaise Pascal, Gottfried Leibniz, 17ος αιώνας)
- Αναλυτική μηχανή του Charles Babbage με πρώτη προγραμματίστρια την Ada Lovelace (19ος αιώνας)
- Alan Turing, θεμελιωτής των εννοιών των αλγορίθμων και του υπολογισμού, η μηχανή Turing (1935)
- MARK I, ο πρώτος ηλεκτρομηχανικός υπολογιστής μεγάλης κλίμακας, IBM, Harvard University (1943)
- John von Neumann, θεμελιωτής της τρέχουσας αρχιτεκτονικής των υπολογιστών (1945)
- ENIAC, ο πρώτος υπολογιστής με λυχνίες κενού, University of Pennsylvania (1946)
- IBM 7090, ο πρώτος υπολογιστής με τρανζίστορ (1961)
- IBM 360, ο πρώτος υπολογιστής με ολοκληρωμένα κυκλώματα ή τσιπ (1964)
- Ο προσωπικός υπολογιστής, το PC (1981)
- Tim Berners-Lee, εισήγαγε την ιδέα του παγκόσμιου ιστού στο διαδίκτυο (1990)
- Ανάπτυξη και διάδοση της κινητής τηλεφωνίας (τελευταία δεκαετία του 20ού αιώνα)

Κάποια υπολογιστικά προβλήματα

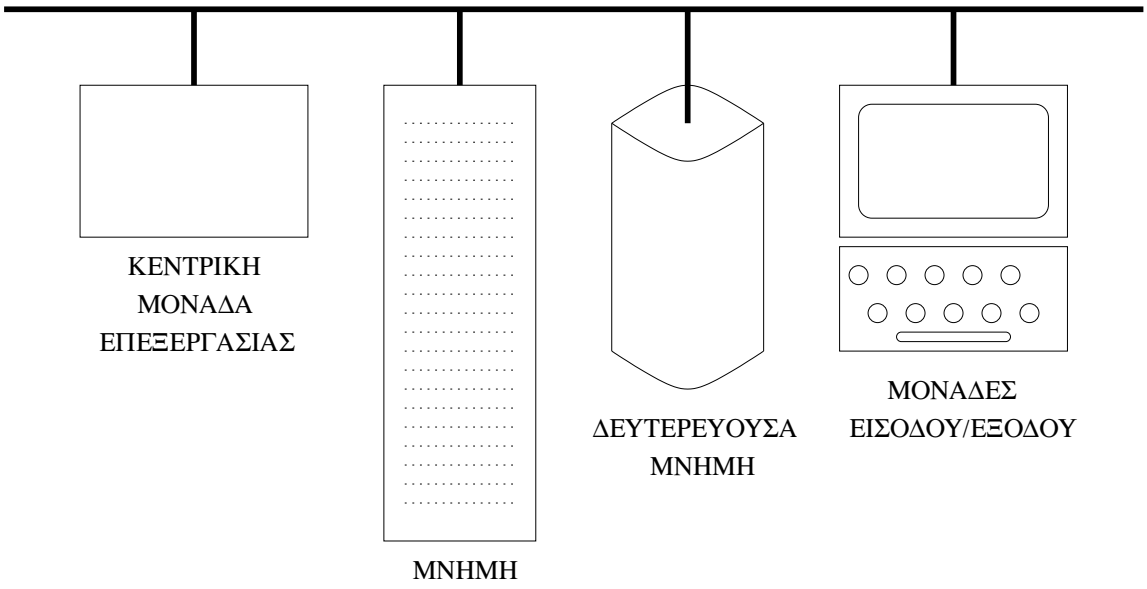
- Πώς προέκυψαν τα αποτελέσματα των πανελληνίων εξετάσεων για την εισαγωγή στα Πανεπιστήμια;
- Πώς αποφασίζεται αν ένας φοιτητής του Τμήματός μας έχει εκπληρώσει τις υποχρεώσεις του για λήψη του πτυχίου του;
- Τι φόρο θα πληρώσουμε φέτος;
- Πώς υπολογίστηκε το τελικό ποσό πληρωμής στους τελευταίους λογαριασμούς τηλεφωνίας, ενέργειας και ύδρευσης που λάβαμε;
- Δεδομένων των αναλυτικών αποτελεσμάτων σε κάθε εκλογικό τμήμα στις εθνικές εκλογές, πόσους βουλευτές εκλέγει το κάθε κόμμα σε κάθε νομό;
- Δεδομένων των αποτελεσμάτων μίας δημοσκόπησης εξόδου (exit poll) σε επιλεγμένα εκλογικά τμήματα, πώς μπορούμε να προβλέψουμε το τελικό ποσοστό ψήφων σε όλη τη χώρα για κάθε κόμμα που συμμετέχει στις εθνικές εκλογές;
- Πώς μπορούμε να προβλέψουμε τον καιρό στην Αθήνα για αύριο ή για τις ερχόμενες ημέρες, δεδομένων όλων των απαιτούμενων μετεωρολογικών στοιχείων;
- Πώς γίνονται οι κρατήσεις αεροπορικών εισιτηρίων;
- Πώς μπορούμε να κατασκευάσουμε το καλύτερο ωρολόγιο πρόγραμμα για τα μαθήματα του τρέχοντος εξαμήνου;
- Το καλύτερο πρόγραμμα εξετάσεων για την ερχόμενη εξεταστική περίοδο;

Πώς να μάθουμε να προγραμματίζουμε;

- Δυστυχώς ο προγραμματισμός δεν διδάσκεται.
- Δεν μπορούμε να μάθουμε να προγραμματίζουμε διαβάζοντας βιβλία για προγραμματισμό.
- Δεν μπορούμε να μάθουμε να προγραμματίζουμε γράφοντας προγράμματα στο χαρτί.
- Ο μόνος τρόπος να μάθουμε να προγραμματίζουμε είναι να γράφουμε προγράμματα στον υπολογιστή, να τα εκτελούμε, να βρίσκουμε τα λάθη που σίγουρα έχουμε κάνει, να τα διορθώνουμε, να εκτελούμε πάλι τα προγράμματα, να ξαναβρίσκουμε λάθη, να τα διορθώνουμε πάλι, μέχρι να καταφέρουμε να κάνουμε τα προγράμματά μας να επιλύουν τα προβλήματά μας όπως πρέπει.
- Επικουρικά, η ανάγνωση και κατανόηση καλογραμμένων, τεκμηριωμένων και ορθών προγραμμάτων είναι πάντοτε χρήσιμη, αλλά από μόνη της δεν αρκεί.

Η δομή του υπολογιστή

- *Κεντρική μονάδα επεξεργασίας*, ο επεξεργαστής
- *Μνήμη*, για προσωρινή αποθήκευση δεδομένων και προγραμμάτων
- *Δευτερεύουσα μνήμη*, για μόνιμη αποθήκευση δεδομένων και προγραμμάτων (δίσκοι, δισκέτες, ταινίες, CDs, DVDs, κλπ.)
- *Μονάδες εισόδου/εξόδου*, για επικοινωνία με τον έξω κόσμο (πληκτρολόγιο, οθόνη, εκτυπωτής, κλπ.)



Η πληροφορία στον υπολογιστή

- Δυαδικά ψηφία 0 και 1 (bits - **b**inary dig**its**)
- Bytes, 1 byte = 8 bits
- Σύμβολα, ASCII κωδικοί (`'A'` = 65, `'B'` = 66, ...)
- Συστήματα αρίθμησης
  - δυαδικό
  - δεκαδικό
  - οκταδικό
  - δεκαεξαδικό

  $(01101101)_2 = (109)_{10} = (155)_8 = (6D)_{16}$
  $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 109$
- Η μνήμη είναι οργανωμένη σε συνεχή κελιά, που το καθένα χαρακτηρίζεται από τη *διεύθυνσή* του, και μέσα στα οποία ο επεξεργαστής μπορεί να καταχωρήσει, αλλά και να διαβάσει απ΄ αυτά, πληροφορίες (δυαδικούς αριθμούς).
- Στη δευτερεύουσα μνήμη, η οργάνωση στο χαμηλό επίπεδο είναι περίπου όπως και στην (κύρια) μνήμη, αλλά οι καταχωρημένες πληροφορίες προσπελαύνονται από τον χρήστη μέσω *αρχείων* (files) και *καταλόγων* (directories), ή, κατά μία ορολογία, *φακέλων* (folders).
- Μονάδες μέτρησης χωρητικότητας μνήμης
  - 1 Kilobyte (KB) = 2^{10} (= $1024 \simeq 10^3$) bytes
  - 1 Megabyte (MB) = 2^{20} (= $1048576 \simeq 10^6$) bytes
  - 1 Gigabyte (GB) = 2^{30} (= $1073741824 \simeq 10^9$) bytes
  - 1 Terabyte (TB) = 2^{40} (= $1099511627776 \simeq 10^{12}$) bytes

Λογισμικό και γλώσσες προγραμματισμού

- Λογισμικό συστήματος
  - Λειτουργικό σύστημα (Microsoft Windows, Linux, macOS ...)
  - Μεταγλωττιστές (`gcc`, `g++`, ...)
  - Κειμενογράφοι (`nano/pico`, `vim`, ...)
  - ...
- Λογισμικό εφαρμογών
- Γλώσσες προγραμματισμού χαμηλού επιπέδου
  - Γλώσσα μηχανής
  - Assembly
- Γλώσσες προγραμματισμού υψηλού επιπέδου
  - Cobol, Fortran, Algol, Ada
  - Pascal, C, Go
  - C++, Java, C#
  - Visual Basic
  - SQL
  - Prolog
  - Haskell, Common Lisp, Scala, F#
  - PHP, Javascript
  - Perl, Python, Ruby
  - ...

Τι χρειαζόμαστε από μία γλώσσα προγραμματισμού;

- Μηχανισμούς για είσοδο (δεδομένων) και έξοδο (αποτελεσμάτων)
  Παράδειγμα:

  Διάβασε *Μισθό Υπαλλήλου*
  Υπολόγισε Κρατήσεις
  Υπολόγισε Καθαρό Μισθό
  Εκτύπωσε *Καθαρό Μισθό*

- Διατύπωση ακολουθίας βημάτων προς εκτέλεση
  Παράδειγμα:

  Έστω Ένα Ορθογώνιο Τρίγωνο ΑΒΓ
  με Ορθή Γωνία στο Α
  Βήμα 1*: Υπολόγισε το Τετράγωνο της ΑΒ*
  Βήμα 2*: Υπολόγισε το Τετράγωνο της ΑΓ*
  Βήμα 3*: Πρόσθεσε τα Δύο Τετράγωνα*
  Βήμα 4*: Η Πλευρά ΒΓ Ισούται με την*
  Τετραγωνική Ρίζα του Αποτελέσματος

- Φύλαξη πληροφοριών στη μνήμη και ανάκτησή τους
  Παράδειγμα:

  Μισθός = 1500
  Αύξηση = 0.04
  *Νέος Μισθός = Αύξηση * Μισθός + Μισθός*

- Διαφοροποίηση ροής του ελέγχου σύμφωνα με συνθήκη
  Παράδειγμα:

  Αν *Κατανάλωση < 200*
  Τότε *Κόστος = 20*
  Αλλιώς *Κόστος = 0.3 * Κατανάλωση - 40*

- Δυνατότητα διατύπωσης επαναληπτικών διαδικασιών
  Παράδειγμα:

  Αριθμός = 1
  Άθροισμα = 0
  Ενόσω *Αριθμός < 1001*
  Άθροισμα = Άθροισμα + Αριθμός^2
  Αριθμός = Αριθμός + 1

- Πακετάρισμα συχνά χρησιμοποιούμενων λειτουργιών
  Παράδειγμα:

  Ταξινόμησε*(Πίνακας Α)*
  Ταξινόμησε*(Πίνακας Β)*
  Συγχώνευσε*(Πίνακας Α, Πίνακας Β, Πίνακας Γ)*

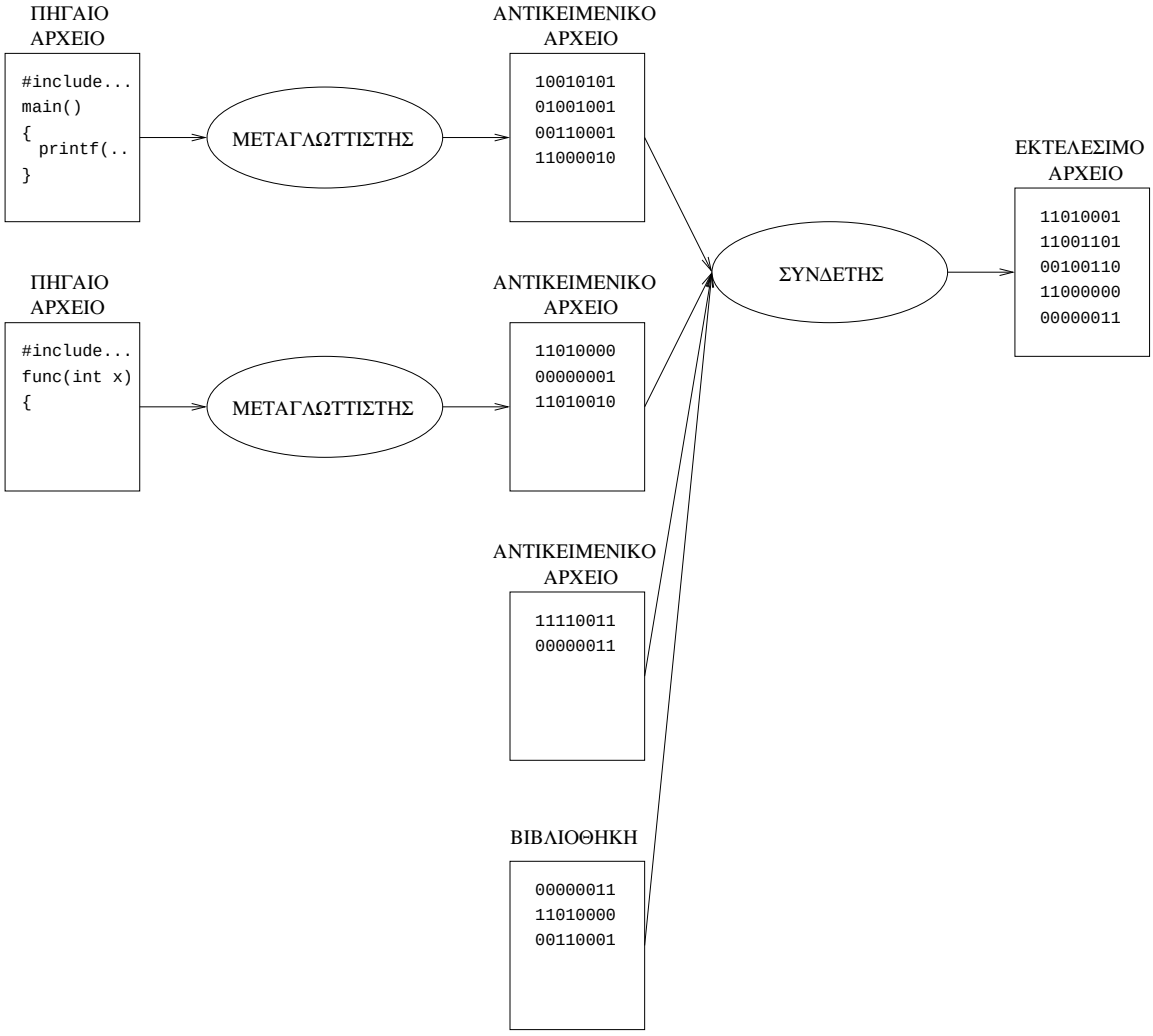
- Μεταφορά της ροής του ελέγχου αλλού (**πολύ σπάνια**)
  Παράδειγμα:

  Βήμα 1*: Αν Χ = 30*
  Τότε ***Πήγαινε*** *στο Βήμα 2*
  Αλλιώς ***Πήγαινε*** *στο Βήμα 4*
  Βήμα 2*: Εκτύπωσε "Τριάντα"*
  Βήμα 3*:* ***Πήγαινε*** *στο Βήμα 5*
  Βήμα 4*: Εκτύπωσε "Οχι τριάντα"*
  Βήμα 5*: ...*

Πώς κατασκευάζουμε εκτελέσιμο πρόγραμμα;

- Συνήθως, γράφουμε τα προγράμματά μας σε κάποια *γλώσσα προγραμματισμού υψηλού επιπέδου*, για παράδειγμα C.
- Για λόγους εύκολης συντήρησης ενός προγράμματος, είναι πιθανόν να το έχουμε διασπάσει σε πολλά *πηγαία αρχεία* (source files), που περιλαμβάνουν *κώδικα* (code) γραμμένο στη γλώσσα προγραμματισμού που έχουμε επιλέξει.
- Ο *μεταγλωττιστής* (compiler) είναι ένα πρόγραμμα που μετατρέπει τα πηγαία αρχεία σε *αντικειμενικά αρχεία* (object files), τα οποία είναι διατυπωμένα στη γλώσσα μηχανής του επεξεργαστή του υπολογιστή μας.
- Τα αντικειμενικά αρχεία δεν είναι άμεσα εκτελέσιμα από τον επεξεργαστή, εκτός του ότι το καθένα απ΄ αυτά μπορεί να μην συνιστά ένα πλήρες πρόγραμμα.
- Ο *συνδέτης* (linker) μετατρέπει ένα σύνολο από αντικειμενικά αρχεία, καθώς και τυχόν *βιβλιοθήκες* (libraries) που θα του δοθούν, σ΄ ένα *εκτελέσιμο αρχείο* (executable file), το οποίο επίσης είναι διατυπωμένο σε γλώσσα μηχανής, αλλά είναι άμεσα εκτελέσιμο από τον επεξεργαστή.

Η διαδικασία της μεταγλώττισης και σύνδεσης



Προγραμματιστικά περιβάλλοντα για την C

- Μεταγλωττιστής `gcc` της C, από το πρόγραμμα ελεύθερου λογισμικού GNU, σε λειτουργικό σύστημα Linux ή macOS
- Περιβάλλοντα που υποστηρίζουν τον μεταγλωττιστή `gcc`, κάτω από λειτουργικό σύστημα Microsoft Windows
  - `WSL` (σε Windows 10/11)
  - `Dev-C++`
  - `VS Code`
  - `Cygwin`
  - `MinGW`
- Περιβάλλον Microsoft Visual Studio για Windows, αλλά **δεν θα χρησιμοποιηθεί στα πλαίσια του παρόντος μαθήματος**

Παραδείγματα χρήσης του gcc

- `gcc myprog.c`
  Μεταγλώττιση του πηγαίου προγράμματος `myprog.c` σε αντικειμενικό αρχείο και κλήση του συνδέτη για την κατασκευή του εκτελέσιμου προγράμματος `a.out`
- `gcc -o myprog myprog.c`
  Μεταγλώττιση του πηγαίου προγράμματος `myprog.c` σε αντικειμενικό αρχείο και κλήση του συνδέτη για την κατασκευή του εκτελέσιμου προγράμματος `myprog`
- `gcc -c myprog.c`
  Μόνο μεταγλώττιση του πηγαίου προγράμματος `myprog.c` στο αντικειμενικό αρχείο `myprog.o`
- `gcc -o prog myprog1.o myprog2.o -lm`
  Μόνο κλήση του συνδέτη για κατασκευή του εκτελέσιμου προγράμματος `prog` από τα αντικειμενικά αρχεία `myprog1.o` και `myprog2.o` και τη μαθηματική βιβλιοθήκη (`m`)
- Άλλες ενδιαφέρουσες επιλογές του `gcc`, εκτός από τις `-o`, `-c` και `-l`:
  - Για να κληθεί μόνο ο προεπεξεργαστής, `-E`
  - Για να παραχθεί το αποτέλεσμα σε γλώσσα assembly, `-S`

Η γλώσσα προγραμματισμού C

- Μεταβλητές, σταθερές, τύποι και δηλώσεις
- Εντολές αντικατάστασης, τελεστές και παραστάσεις
- Η ροή του ελέγχου
- Δομή προγράμματος, συναρτήσεις και εξωτερικές μεταβλητές
- Εμβέλεια και χρόνος ζωής μεταβλητών
- Αναδρομή
- Διευθύνσεις θέσεων μνήμης, δείκτες και πίνακες
- Δυναμική δέσμευση μνήμης
- Συμβολοσειρές
- Πίνακες δεικτών, δείκτες σε δείκτες και πολυδιάστατοι πίνακες
- Δείκτες σε συναρτήσεις
- Ορίσματα γραμμής εντολών
- Απαριθμήσεις, δομές, αυτο-αναφορικές δομές (λίστες, δυαδικά δέντρα), ενώσεις, πεδία bit και δημιουργία νέων ονομάτων τύπων
- Είσοδος και έξοδος
- Χειρισμός αρχείων
- Προεπεξεργαστής της C και μακροεντολές

Καλημέρα κόσμε της C

```
/* File: helloworld.c */
#include <stdio.h>

int main()
{ printf("Hello world\n");
}
```

```
% gcc -o helloworld helloworld.c
% ./helloworld
Hello world
%
```

- Ό,τι περικλείεται μέσα σε `/*` και `*/` θεωρείται σχόλιο και δεν λαμβάνεται υπόψη από τον μεταγλωττιστή.
- Η γραμμή `#include <stdio.h>` είναι οδηγία προς τον προεπεξεργαστή της C να συμπεριλάβει σ΄ εκείνο το σημείο τα περιεχόμενα του *αρχείου επικεφαλίδας* (header file) `stdio.h`, το οποίο περιέχει χρήσιμες δηλώσεις για τις συναρτήσεις εισόδου/εξόδου.
- Κάθε πρόγραμμα C πρέπει να περιέχει ακριβώς μία συνάρτηση με όνομα `main`, που είναι αυτή από την οποία θα αρχίσει να εκτελείται το εκτελέσιμο πρόγραμμα που θα προκύψει από τη μεταγλώττιση. Για το `int` στον ορισμό της `main`, θα γίνει συζήτηση σε επόμενη φάση.
- Η `printf` είναι μία συνάρτηση εξόδου, που όταν κληθεί, εκτυπώνει ό,τι της έχει δοθεί μέσα στις παρενθέσεις, στην προκείμενη περίπτωση το *αλφαριθμητικό ή συμβολοσειρά* (string) `"Hello world"` (χωρίς τα "), ακολουθούμενο από μία αλλαγή γραμμής (`\n`).

Πόσο είναι το π;

```
/* File: picomp.c */
#include <stdio.h>            /* Header file for standard I/O library */
#include <math.h>                     /* Header file for math library */

int main()
{ long i;
  double sum, current, pi;
  i = 1;                               /* Denominator of current term */
  sum = 0.0;                                          /* So far sum */
  do {
    current = 1/(((double) i)*((double) i));          /* Current term */
    sum = sum+current;                    /* Add current term to sum */
    i++;                                             /* Next term now */
  } while (current > 1.0e-15);  /* Stop if current term is very small */
  pi = sqrt(6*sum);                     /* Compute approximation of pi */
  printf("Summed %8ld terms, pi is %10.8f\n", i-1, pi);
}
```

```
% gcc -o picomp picomp.c -lm
% ./picomp
Summed 31622777 terms, pi is 3.14159262
%
```

- Το πρόγραμμα υπολογίζει το π με βάση τη σειρά:

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \frac{1}{6^2} + \frac{1}{7^2} + \frac{1}{8^2} + \ldots$$

- Το `#include <math.h>` χρειάζεται επειδή χρησιμοποιείται στο πρόγραμμα η μαθηματική συνάρτηση `sqrt`, για την εύρεση τετραγωνικής ρίζας. Προσοχή: Στην εντολή μεταγλώττισης και σύνδεσης (`gcc`) χρειάζεται και το `-lm`, για να συμπεριληφθεί στο εκτελέσιμο πρόγραμμα και η μαθηματική βιβλιοθήκη της C.

- Το `i` συμβολίζει μία ακέραια μεταβλητή στην οποία θα φυλάσσεται, δηλαδή στην περιοχή της μνήμης που της αντιστοιχεί ο μεταγλωττιστής, η τιμή του τρέχοντος παρονομαστή κατά τον υπολογισμό του αθροίσματος. Δηλώνεται με μέγεθος `long`, ώστε να μπορούν να φυλαχθούν σ΄ αυτήν ακέραιοι μεγέθους τουλάχιστον 4 (ή 8) bytes, ήτοι τουλάχιστον μέχρι το $2^{31} - 1 = 2147483647$ (ή το $2^{63} - 1 = 9223372036854775807$). Αν είχε δηλωθεί απλώς σαν `int`, μπορεί να περιοριζόμασταν σε 2 bytes για τη φύλαξη, δηλαδή έως το $2^{15} - 1 = 32767$, αν και στους σύγχρονους υπολογιστές, οι `int` είναι συνήθως των 4 bytes.

- Τα `sum`, `current` και `pi` είναι μεταβλητές για τη φύλαξη πραγματικών αριθμών, του μέχρι στιγμής υπολογισμένου αθροίσματος, του τρέχοντος όρου της σειράς και της τελικής προσεγγιστικής τιμής του π, αντίστοιχα. Δηλώθηκαν σαν `double` για να έχουμε εσωτερική αναπαράσταση των αριθμών αυτών με μεγαλύτερη ακρίβεια (διπλή) σε δεκαδικά ψηφία, απ΄ αυτήν που θα είχαμε αν είχαν δηλωθεί σαν `float`.

- Με τις εντολές αντικατάστασης `i = 1` και `sum = 0.0`, δίνουμε αρχικές τιμές στον τρέχοντα παρονομαστή και στο μέχρι στιγμής άθροισμα.

- Η ομάδα εντολών μέσα στο μπλοκ `do {.....} while` θα εκτελείται συνεχώς, όσο ο τρέχων όρος του αθροίσματος είναι μεγαλύτερος από κάποια ελάχιστη τιμή, την 10^{-15} (= `1.0e-15`).

- Σε κάθε επανάληψη του μπλοκ `do {.....} while`, υπολογίζουμε αρχικά τον τρέχοντα όρο του αθροίσματος (το πρόθεμα `(double)` στο `i` υπάρχει για να μετατραπεί η τιμή του ακεραίου `i` σε πραγματική τιμή διπλής ακρίβειας, πριν συμμετάσχει στην αριθμητική πράξη). Μετά προστίθεται ο τρέχων όρος του αθροίσματος στο μέχρι στιγμής άθροισμα, με το `sum = sum+current` (συνήθως το γράφουμε και σαν `sum += current`), και τέλος αυξάνουμε τον τρέχοντα παρονομαστή κατά 1, με το `i++` (ισοδύναμο με το `i = i+1`).
- Όταν τελειώσει το μπλοκ `do {.....} while`, στη μεταβλητή `sum` υπάρχει το άθροισμα όλων των όρων της σειράς που είναι μεγαλύτεροι από 10^{-15}. Αυτό είναι περίπου ίσο με το $\frac{\pi^2}{6}$. Οπότε, μετά υπολογίζουμε προσεγγιστικά το π με την εντολή `pi = sqrt(6*sum)` (δηλαδή $\sqrt{6 \times sum}$).
- Με την κλήση της συνάρτησης `printf` εκτυπώνεται το αποτέλεσμα. Συγκεκριμένα, εκτυπώνεται ό,τι υπάρχει μέσα στα ", αλλά στη θέση του `%8ld` εκτυπώνεται η τιμή του `i-1`, σαν δεκαδικός (`d`) ακέραιος μεγέθους `long` (`l`) σε 8 τουλάχιστον θέσεις, στοιχισμένος δεξιά. Επίσης, στη θέση του `%10.8f` εκτυπώνεται η υπολογισμένη τιμή του π, σαν πραγματικός αριθμός (`f`) σε 10 τουλάχιστον θέσεις, με δεξιά στοίχιση, από τις οποίες οι 8 για το κλασματικό μέρος.

Γράψτε προγράμματα C που να υπολογίζουν με αρκετή ακρίβεια τα παρακάτω αθροίσματα. Μπορείτε να βρείτε με ποιους ενδιαφέροντες αριθμούς ισούνται;

$$S_1 = \frac{1}{1^2} - \frac{1}{2^2} + \frac{1}{3^2} - \frac{1}{4^2} + \frac{1}{5^2} - \frac{1}{6^2} + \frac{1}{7^2} - \frac{1}{8^2} + \dots$$

$$S_2 = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \frac{1}{9} - \dots$$

$$S_3 = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \frac{1}{17} - \dots$$

$$S_4 = \frac{1}{1 \times 3} + \frac{1}{3 \times 5} + \frac{1}{5 \times 7} + \frac{1}{7 \times 9} + \frac{1}{9 \times 11} + \dots$$

$$S_5 = \frac{1}{2 \times 3 \times 4} - \frac{1}{4 \times 5 \times 6} + \frac{1}{6 \times 7 \times 8} - \frac{1}{8 \times 9 \times 10} + \dots$$

$$S_6 = \frac{1}{1^4} + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \frac{1}{5^4} + \frac{1}{6^4} + \frac{1}{7^4} + \frac{1}{8^4} + \dots$$

Το ίδιο και για το γινόμενο:

$$P = \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \frac{8}{9} \dots$$

Πότε είναι το Ορθόδοξο Πάσχα;

```
/* File: easter.c */
#include <stdio.h>
#define STARTYEAR 2010
#define ENDYEAR   2025

int main()
{ int year, a, b, c, d, e;
  for (year = STARTYEAR ; year <= ENDYEAR ; year++) {
    a = year % 19;
    b = year % 4;    /* Gauss method for Easter date computation */
    c = year % 7;
    d = (19*a+15) % 30;
    e = (2*b+4*c+6*d+6) % 7;
    printf("Easter in the year %d: ", year);
    if (d+e+4 > 30)                              /* Easter in May */
      printf("  May %d\n", d+e-26);
    else                                       /* Easter in April */
      printf("April %d\n", d+e+4);
  }
}
```

```
% gcc -o easter easter.c
% ./easter
Easter in the year 2010: April 4
Easter in the year 2011: April 24
Easter in the year 2012: April 15
Easter in the year 2013:   May 5
Easter in the year 2014: April 20
Easter in the year 2015: April 12
Easter in the year 2016:   May 1
Easter in the year 2017: April 16
Easter in the year 2018: April 8
Easter in the year 2019: April 28
Easter in the year 2020: April 19
Easter in the year 2021:   May 2
Easter in the year 2022: April 24
Easter in the year 2023: April 16
Easter in the year 2024:   May 5
Easter in the year 2025: April 20
%
```

- Το πρόγραμμα εφαρμόζει τη μέθοδο του Gauss για τον υπολογισμό της ημερομηνίας του Πάσχα το έτος $year$. Σύμφωνα με αυτήν, το Πάσχα είναι στις $(d + e + 4)$ Απριλίου (αν η ημερομηνία αυτή είναι μεγαλύτερη από 30, τότε πέφτει μέσα στον Μάιο), όπου $d = (19 \cdot a + 15) \bmod 30$, $e = (2 \cdot b + 4 \cdot c + 6 \cdot d + 6) \bmod 7$, $a = year \bmod 19$, $b = year \bmod 4$ και $c = year \bmod 7$.

- Με τις δηλώσεις `#define .....` ζητάμε από τον προεπεξεργαστή της C να αντικαταστήσει μέσα στο αρχείο, όπου βρει το σύμβολο που ακολουθεί το `#define` με το σύμβολο που είναι αμέσως μετά. Στην προκείμενη περίπτωση, ζητάμε να αντικατασταθεί το `STARTYEAR` με 2000 και το `ENDYEAR` με 2015.

- Το μπλοκ `for (` E_1 `;` E_2 `;` E_3 `) {.....}` περιγράφει έναν βρόχο που θα πρέπει να εκτελεσθεί επαναλαμβανόμενα. Πριν αρχίσει η εκτέλεσή του, θα εκτελεσθεί η εντολή E_1. Εφ΄ όσον η συνθήκη E_2 είναι αληθινή, θα γίνει η τρέχουσα επανάληψη στο βρόχο. Με το τέλος κάθε επανάληψης, εκτελείται και η εντολή E_3.

- Στις εντολές αντικατάστασης, ο τελεστής `%` παριστάνει το υπόλοιπο διαίρεσης (mod).

- Η δομή `if (` C `)` E_1 `else` E_2 σημαίνει ότι θέλουμε να εκτελεσθεί η εντολή (ή μπλοκ εντολών) E_1, αν η συνθήκη C είναι αληθής, ή η εντολή (ή μπλοκ εντολών) E_2, αν η συνθήκη C είναι ψευδής.

Να κατασκευασθεί μαγικό τετράγωνο 5×5

```
/* File: magic.c */
#include <stdio.h>
#define SIZE 5

int main()
{ int i, j, k;
  int x[SIZE][SIZE];                        /* The board to be filled */
  for (i=0 ; i < SIZE ; i++)
    for (j=0 ; j < SIZE ; j++)
      x[i][j] = 0;                       /* Mark all squares as empty */
  i = (SIZE-1)/2;                                  /* Go to middle */
  j = SIZE-1;                                      /* right square */
  for (k=1 ; k <= SIZE*SIZE ; k++) {
    if (x[i][j] != 0) {                     /* If current not empty */
      i = i-1;                                /* Move one square up */
      j = j-2;                               /* and two squares left */
      if (i < 0) i = i+SIZE;                  /* If you are outside */
      if (j < 0) j = j+SIZE;  /* go to the respective square inside */
    }
    x[i][j] = k;                /* Fill current square with number k */
    i++;                                     /* Move one square down */
    if (i == SIZE) i = 0;                   /* If outside get inside */
    j++;                                    /* Move one square right */
    if (j == SIZE) j = 0;                   /* If outside get inside */
  }
  for (i=0 ; i < SIZE ; i++) {
    for (j=0 ; j < SIZE ; j++)
      printf("%4d ", x[i][j]);                   /* Print out board */
    printf("\n");
  }
}
```

```
% gcc -o magic magic.c
% ./magic
  11   10    4   23   17
  18   12    6    5   24
  25   19   13    7    1
   2   21   20   14    8
   9    3   22   16   15
%
```

- Το πρόγραμμα εφαρμόζει τη μέθοδο του De la Loubère για να κατασκευάσει ένα μαγικό τετράγωνο 5×5 (η μέθοδος εφαρμόζεται για περιττό μήκος πλευράς), δηλαδή ένα πλαίσιο με 25 θέσεις σε διάταξη τετραγώνου μέσα στις οποίες είναι τοποθετημένοι οι αριθμοί από το 1 έως το 25, μία φορά ο καθένας, έτσι ώστε σε κάθε γραμμή, κάθε στήλη και κάθε μία από τις δύο διαγωνίους να έχουμε το ίδιο άθροισμα (αλήθεια, πόσο;). Για την περιγραφή της μεθόδου σε μία ψευδογλώσσα, δείτε τον αλγόριθμο 118, στη δεύτερη σελίδα του `http://www.di.uoa.gr/~ip/magic-sq.pdf`. Ο αλγόριθμος 117, στην πρώτη σελίδα του κειμένου αυτού, μπορεί να εφαρμοσθεί για την κατασκευή μαγικών τετραγώνων άρτιας πλευράς.

- Με τη δήλωση `int x[SIZE][SIZE]` ζητάμε να δεσμευθεί χώρος για ένα δισδιάστατο πίνακα ακεραίων `x` με `SIZE` ($= 5$) γραμμές και `SIZE` στήλες. Προσοχή: Οι γραμμές και οι στήλες αριθμούνται από το 0 έως το `SIZE-1` ($= 4$). Με την έκφραση `x[3][1]`, μπορούμε να αναφερθούμε στο στοιχείο που βρίσκεται στην 4η γραμμή και 2η στήλη. Φυσικά, στην C μπορούμε να ορίσουμε και μονοδιάστατους πίνακες, αλλά και πίνακες με διάσταση μεγαλύτερη από 2.

- Παρατηρήστε τη χρήση των εντολών `if`. Δεν υπάρχει τμήμα `else`. Αν ισχύει η συνθήκη, εκτελείται η εντολή (ή μπλοκ εντολών) που ακολουθεί. Αλλιώς, ο έλεγχος πηγαίνει στην επόμενη εντολή. Ο τελεστής `!=` στη συνθήκη της πρώτης εντολής `if` σημαίνει "διάφορο" (έλεγχος αν το τρέχον τετραγωνίδιο είναι γεμάτο).

Μετατροπή πεζών γραμμάτων σε κεφαλαία

```
/* File: capitalize.c */
#include <stdio.h>

int main()
{ int ch; /* Be careful! Declare ch as int because of getchar() and EOF */
  ch = getchar();                                  /* Read first character */
  while (ch != EOF) {           /* Go on if we didn't reach end of file */
    if (ch >= 'a' && ch <= 'z')                    /* If lower case letter */
      ch = ch - ('a'-'A'); /* Move 'a'-'A' positions in the ASCII table */
    putchar(ch);                                    /* Print out character */
    ch = getchar();                                 /* Read next character */
  }
}
```

```
% gcc -o capitalize capitalize.c
% cat capinp.txt
This is a text file with 2 lines to
test the C program "capitalize.c".
% ./capitalize < capinp.txt
THIS IS A TEXT FILE WITH 2 LINES TO
TEST THE C PROGRAM "CAPITALIZE.C".
%
```

- Η συνάρτηση `getchar` παρέχεται από την πρότυπη (standard) βιβλιοθήκη εισόδου/εξόδου της C. Διαβάζει ένα χαρακτήρα από την είσοδο και επιστρέφει στο όνομά της την ακέραια τιμή του ASCII κωδικού του χαρακτήρα. Αν δεν υπάρχει άλλος χαρακτήρας να διαβαστεί, δηλαδή φτάσαμε στο τέλος της εισόδου, τότε επιστρέφει στο όνομά της μία ειδική τιμή, το `EOF`, που είναι `#define`'d σε μία συγκεκριμένη ακέραια τιμή μέσα στο αρχείο επικεφαλίδας `stdio.h`.

- Η ομάδα εντολών μέσα στο μπλοκ `while` {.....} θα εκτελείται συνεχώς, όσο υπάρχουν και άλλοι χαρακτήρες για να διαβάσουμε από την είσοδο.
- Η συνθήκη μέσα στο `if` είναι μία λογική σύζευξη (τελεστής `&&`). Τα `'a'` και `'A'` είναι σταθερές χαρακτήρα και ισούνται με την αριθμητική τιμή των κωδικών των αντίστοιχων χαρακτήρων στον ASCII πίνακα, δηλαδή `'a'`=97 και `'A'`=65.
- Το συγκεκριμένο πρόγραμμα βασίζεται στην υπόθεση ότι στον ASCII πίνακα οι κεφαλαίοι λατινικοί χαρακτήρες είναι σε συνεχόμενες θέσεις και ότι το ίδιο ισχύει και για τους πεζούς. Αυτό όντως ισχύει, γι΄ αυτό και το πρόγραμμα δουλεύει σωστά. Για την μετατροπή ενός πεζού χαρακτήρα σε κεφαλαίο, δεν χρειάζεται να γνωρίζουμε τις ακριβείς τιμές των `ASCII` κωδικών, αρκεί να μετατοπίσουμε τον πεζό χαρακτήρα κατά `'a'-'A'` θέσεις.
- Η συνάρτηση `putchar` παρέχεται επίσης από την πρότυπη βιβλιοθήκη εισόδου/εξόδου της C (είναι η "αδελφή" συνάρτηση της `getchar`) και η λειτουργία της είναι να εκτυπώσει στην έξοδο τον χαρακτήρα του οποίου τον ASCII κωδικό δίνουμε μέσα στις παρενθέσεις.

Μεταβλητές, σταθερές, τύποι και δηλώσεις στην C

- Μεταβλητές
  - Συμβολικά ονόματα για θέσεις μνήμης
  - Αποθήκευση και ανάκτηση δεδομένων
  - Ανήκουν σε τύπους
- Σταθερές
  - Συγκεκριμένες τιμές
  - Ανήκουν σε τύπους
- Τύποι δεδομένων
  - Για ακεραίους
    - `short int` ή `short` (συνήθως 2 bytes)
    - `int` (συνήθως 4, σπανιότερα 2 ή 8, bytes, ανάλογα με το μέγεθος λέξης του επεξεργαστή)
    - `long int` ή `long` (συνήθως 4, μερικές φορές 8, bytes)
    - `long long int` ή `long long` (συνήθως 8 bytes)
  - Για χαρακτήρες, αλλά και ακεραίους
    - `char` (1 byte)
  - Για πραγματικούς αριθμούς (κινητής υποδιαστολής)
    - `float` (απλής ακρίβειας, συνήθως 4 bytes)
    - `double` (διπλής ακρίβειας, συνήθως 8 bytes)
    - `long double` (εκτεταμένης ακρίβειας, συνήθως 16 bytes)

- Παραδείγματα σταθερών
  - Ακέραιες σταθερές (θεωρούνται τύπου `int`):
    `237`, `-12345`, `22431L` (το `L` συμβολίζει ότι επιθυμούμε να θεωρηθεί η σταθερά τύπου `long`), `0224` (ο οκταδικός αριθμός 224), `0x1C` (ο δεκαεξαδικός αριθμός `1C`)
  - Σταθερές χαρακτήρα:
    `'A'` (ο ASCII κωδικός του χαρακτήρα `A`, δηλαδή 65)
    `'\n'` (ο ASCII κωδικός για την αλλαγή γραμμής)
    `'\t'` (ο ASCII κωδικός για τον στηλογνώμονα, tab)
    `'0'` (ο ASCII κωδικός του χαρακτήρα `0`, δηλαδή 48)
    `'\0'` (ο χαρακτήρας με ASCII κωδικό 0)
    `'\145'` (ο χαρακτήρας με ASCII κωδικό τον οκταδικό αριθμό `145`)
    `'\xA2'` (ο χαρακτήρας με ASCII κωδικό τον δεκαεξαδικό αριθμό `A2`)
  - Σταθερές κινητής υποδιαστολής (θεωρούνται τύπου `double`):
    `23.78`, `-1.24e-14` (δηλαδή $-1.24 \cdot 10^{-14}$)
- Μία ειδική κατηγορία σταθερών είναι τα αλφαριθμητικά ή συμβολοσειρές (strings). Μία συμβολοσειρά είναι μία ακολουθία από χαρακτήρες μέσα σε ", για παράδειγμα `"Hello world\n"`. Πρόκειται, ουσιαστικά, για έναν πίνακα χαρακτήρων, χωρίς τα ", με τελευταίο στοιχείο του πίνακα το `'\0'` (παρόλο που δεν σημειώνεται στη διατύπωση της συμβολοσειράς μέσα στα ").

- Με μία δήλωση ενημερώνεται ο μεταγλωττιστής για την πρόθεσή μας να χρησιμοποιήσουμε στο πρόγραμμά μας κάποια ή κάποιες μεταβλητές συγκεκριμένου τύπου, τις οποίες μπορούμε να αρχικοποιήσουμε και με σταθερές του τύπου αυτού, αν θέλουμε, και αντιστοιχεί σ΄ αυτές κατάλληλες θέσεις μνήμης. Παραδείγματα:

```
int i, j, k;
char ch;
float x, y = 2.34;
long count, cost;
double pi = 3.14159;
```

- Τα ονόματα των μεταβλητών μπορούν να περιλαμβάνουν, μέχρι 31 συνολικά, πεζούς και κεφαλαίους χαρακτήρες, αριθμούς και τον χαρακτήρα _.

- Δεν επιτρέπεται η χρήση δεσμευμένων λέξεων της C (π.χ. `if`, `float`, `for`, κλπ.) για ονόματα μεταβλητών.

- Δεν είναι καλό να ορίζουμε μεταβλητές στα προγράμματά μας που αρχίζουν από _, γιατί τέτοια ονόματα χρησιμοποιούνται για τις μεταβλητές από τις βιβλιοθήκες και ενδέχεται να δημιουργηθεί πρόβλημα.

- Παρότι είναι επιτρεπτό, δεν είναι καλό ονόματα μεταβλητών να αποτελούνται μόνο από κεφαλαίους χαρακτήρες, γιατί συνήθως δίνουμε τέτοια ονόματα στις συμβολικές σταθερές που ορίζουμε με `#define`, για να τις διαχειριστεί ο προεπεξεργαστής.

- Στη δήλωση μίας ακέραιας μεταβλητής ή μίας μεταβλητής χαρακτήρα, μπορεί να προηγηθεί ο προσδιοριστής `unsigned`, που δηλώνει ότι οι αριθμοί που φυλάσσονται στη μεταβλητή πρέπει να ερμηνευθούν σαν θετικοί (ή μηδέν).

- Αν υπάρχει στη δήλωση ο προσδιοριστής `signed`, ή δεν υπάρχει κανένας, το περιεχόμενο της μεταβλητής θεωρείται ακέραιος με πρόσημο (θετικό ή αρνητικό).

- Παραδείγματα:

```
signed char ch;
unsigned char uch;
short i;
unsigned long int k;
```

  - Στη μεταβλητή `ch` φυλάσσονται αριθμοί από -128 $(= -2^7)$ έως 127 $(= 2^7 - 1)$, ενώ στην `uch` από 0 έως 255 $(= 2^8 - 1)$
  - Στη μεταβλητή `i` φυλάσσονται αριθμοί από -32768 $(= -2^{15})$ έως 32767 $(= 2^{15} - 1)$, ενώ στην `k` από 0 έως 4294967295 $(= 2^{32} - 1)$

- Σε δήλωση μεταβλητής, με ταυτόχρονη αρχικοποίηση, μπορεί να εφαρμοσθεί ο προσδιοριστής `const`, που εξασφαλίζει ότι η τιμή της μεταβλητής δεν θα αλλάξει. Παράδειγμα:

```
const double e = 2.71828;
```

- Για μεταβλητές, σταθερές, τύπους και δηλώσεις, αναλυτικότερα στις παραγράφους §2.1 έως §2.4 του [KR]. [α΄]

[α΄] Για τη συνέχεια, ο συμβολισμός [KR] θα αναφέρεται στο βιβλίο: Brian W. Kernighan, Dennis M. Ritchie, "Η Γλώσσα Προγραμματισμού C".

Εντολές αντικατάστασης, τελεστές και παραστάσεις

- Οι εντολές σ΄ ένα πρόγραμμα C, όπως και οι δηλώσεις, τελειώνουν με ένα ελληνικό ερωτηματικό (;).
- Ένα βασικό είδος εντολής είναι η εντολή αντικατάστασης (ή κατά την πιο συνήθη ορολογία, εντολή ανάθεσης). Είναι της μορφής: ⟨μεταβλητή⟩ = ⟨παράσταση⟩
- Μία εντολή αντικατάστασης έχει σαν αποτέλεσμα τον υπολογισμό της τιμής της παράστασης στο δεξί μέλος του = και την ανάθεση της τιμής αυτής στη μεταβλητή στο αριστερό μέλος του =.
- Πολύ συχνές είναι οι αριθμητικές παραστάσεις που περιγράφουν πράξεις μεταξύ ακέραιων ή πραγματικών μεταβλητών και σταθερών, μέσω των αριθμητικών τελεστών, δηλαδή +, −, *, / και % (υπόλοιπο διαίρεσης).
- Προσοχή! Διαίρεση μεταξύ ακεραίων δίνει σαν αποτέλεσμα το πηλίκο της διαίρεσης.

- Παραστάσεις που εμπλέκουν μεταβλητές και σταθερές διαφορετικών τύπων δίνουν σαν αποτέλεσμα τιμή του "ευρύτερου" τύπου. Για παράδειγμα, κάποια αριθμητική πράξη μεταξύ `int` και `double` δίνει αποτέλεσμα `double`.
- Ανάθεση μίας τιμής ενός τύπου σε μεταβλητή άλλου τύπου έχει σαν αποτέλεσμα τη μετατροπή της τιμής στον τύπο της μεταβλητής, είτε χωρίς απώλεια πληροφορίας, είτε με απώλεια πληροφορίας, ανάλογα με το αν ο τύπος της μεταβλητής έχει τη δυνατότητα να αναπαραστήσει πλήρως την ανατιθέμενη τιμή.
- Ρητή μετατροπή τύπου μπορεί να γίνει μέσω προσαρμογής (cast), για παράδειγμα, αν η μεταβλητή `i` είναι ακέραιου τύπου, η έκφραση `(double) i` αναφέρεται στην τιμή της `i` σαν πραγματικό αριθμό διπλής ακρίβειας.

- Δύο πολύ χρήσιμοι τελεστές είναι οι τελεστές μοναδιαίας αύξησης (++) και μείωσης (--). Όταν εφαρμόζονται επάνω σε μία μεταβλητή, είτε στα αριστερά είτε στα δεξιά της, έχουν σαν αποτέλεσμα την αύξηση, ή μείωση αντίστοιχα, της τιμής της μεταβλητής κατά 1. Για παράδειγμα, η ακολουθία εντολών

```
x = 2;
y = 9;
x++;
--y;
++x;
x--;
y--;
++y;
```

  θα έχει σαν τελικό αποτέλεσμα οι μεταβλητές x και y να έχουν τελικά σαν τιμές τις 3 και 8, αντίστοιχα. Στο παράδειγμα αυτό, δεν είχε κάποια σημασία η τοποθέτηση των τελεστών αριστερά ή δεξιά της μεταβλητής.

- Εκτός από αυτόνομη εντολή, η εφαρμογή των τελεστών μοναδιαίας αύξησης σε μία μεταβλητή, μπορεί να παίξει και το ρόλο παράστασης (ή τμήματος παράστασης). Στην περίπτωση αυτή, έχει σημασία αν ο μοναδιαίος τελεστής είναι προθεματικός (αριστερά της μεταβλητής) ή μεταθεματικός (δεξιά της μεταβλητής).

- Ένας προθεματικός τελεστής μοναδιαίας αύξησης ή μείωσης σε μία παράσταση υπονοεί ότι πρώτα πρέπει να γίνει η τροποποίηση της τιμής της μεταβλητής (αύξηση ή μείωση, ανάλογα) και μετά να χρησιμοποιηθεί η τροποποιημένη τιμή για τον υπολογισμό της τιμής της παράστασης.
- Ένας μεταθεματικός τελεστής μοναδιαίας αύξησης ή μείωσης σε μία παράσταση υπονοεί ότι πρώτα πρέπει να χρησιμοποιηθεί η τρέχουσα τιμή της μεταβλητής για τον υπολογισμό της τιμής της παράστασης και μετά να γίνει η τροποποίηση της τιμής της μεταβλητής (αύξηση ή μείωση, ανάλογα).
- Παράδειγμα:

```
x = 4;
y = 7;
z = ++y;
y = z - (x++);
z = x - (--y);
```

  Τι τιμή θα έχουν οι μεταβλητές `x`, `y` και `z` μετά από αυτές τις εντολές; [α΄]

[α΄] 5, 3 και 2, αντίστοιχα

- Πολύ συχνά, στην C, θέλουμε να ελέγξουμε κάποια συνθήκη και ανάλογα με το αποτέλεσμα του ελέγχου (αληθές ή ψευδές) να προβούμε σε κάποια ενέργεια.
- Οι συνθήκες συνήθως είναι συγκρίσεις τιμών αριθμητικών παραστάσεων, μέσω των συσχετιστικών τελεστών (ή τελεστών σύγκρισης), που είναι οι >, <, >=, <=, == (ίσο) και != (διάφορο).
- Πιο σύνθετες συνθήκες μπορούν να κατασκευασθούν από απλούστερες μέσω των λογικών τελεστών && (ΚΑΙ), || (Ή) και ! (ΟΧΙ).
- Ουσιαστικά, οι συνθήκες δεν είναι κάτι διαφορετικό από τις παραστάσεις. Μία αληθής συνθήκη είναι μία παράσταση με τιμή ίση με 1, ενώ μία ψευδής συνθήκη θεωρείται ότι έχει τιμή 0. Αντίστοιχα, μία παράσταση με τιμή διάφορη του 0 μπορεί να παίξει τον ρόλο μίας αληθούς συνθήκης, ενώ αν η τιμή της είναι 0, είναι ισοδύναμη με μία ψευδή συνθήκη. Για παράδειγμα, γράφοντας

  ```
  if (myvar != 0) { ........
  ```

  είναι ακριβώς το ίδιο με το

  ```
  if (myvar) { ........
  ```

- Μερικές φορές, είναι πολύ χρήσιμο να κάνουμε πράξεις σε επίπεδο bit μεταξύ ακεραίων τιμών (μεταβλητών και σταθερών). Αυτό επιτυγχάνεται μέσω των τελεστών πράξεων με bit, που είναι οι:
  - `&`: ΚΑΙ
  - `|`: Ή
  - `^`: αποκλειστικό Ή
  - `<<`: ολίσθηση αριστερά
  - `>>`: ολίσθηση δεξιά
  - `~`: συμπλήρωμα ως προς ένα
- Παράδειγμα:

```
unsigned char x;
x = 178;
x = x & 074;
x = x | 0xD6;
x = x ^ 104;
x = x >> 3;
x = x << 2;
x = ~x;
```

  Ποια θα είναι η τελική τιμή της μεταβλητής `x`; [α΄]

[α΄] 179

- Κάθε διμελής τελεστής (+, −, *, /, %, &, |, ^, <<, >>) μπορεί να χρησιμοποιηθεί και με ένα σύντομο τρόπο σε μία εντολή αντικατάστασης, ως εξής:
  ⟨μεταβλητή⟩ ⟨τελεστής⟩= ⟨παράσταση⟩
  Αυτή η εντολή είναι ισοδύναμη με την:
  ⟨μεταβλητή⟩ = ⟨μεταβλητή⟩ ⟨τελεστής⟩ ⟨παράσταση⟩
  Για παράδειγμα, η εντολή

  ```
  sum += x;
  ```

  είναι ισοδύναμη με την

  ```
  sum = sum + x;
  ```

- Μερικές φορές, είναι χρήσιμες και οι παραστάσεις υπό συνθήκη, σαν την
  ⟨συνθήκη⟩ ? ⟨παράσταση⟩$_1$: ⟨παράσταση⟩$_2$
  Η τιμή αυτής της παράστασης είναι αυτή που έχει η ⟨παράσταση⟩$_1$, αν η ⟨συνθήκη⟩ είναι αληθής (δηλαδή, αν έχει τιμή διάφορη του 0, θεωρώντας την ως παράσταση και αυτή), ή αυτή που έχει η ⟨παράσταση⟩$_2$, αν η ⟨συνθήκη⟩ είναι ψευδής (δηλαδή ίση με 0). Παράδειγμα:

  ```
  z = (a > b) ? a : b;
  ```

  Με την εντολή αυτή, η τιμή της z θα γίνει ίση με τη μεγαλύτερη τιμή από τις a και b.

- Η χρήση διαφόρων τελεστών μέσα σε μία παράσταση μπορεί να προκαλέσει σύγχυση στον αναγνώστη ενός προγράμματος για το πώς ακριβώς θα πρέπει να υπολογισθεί η τιμή της παράστασης, δηλαδή για τη σειρά με την οποία θα πρέπει να εφαρμοσθούν οι τελεστές, όχι όμως και στον μεταγλωττιστή.
- Υπάρχουν συγκεκριμένες προτεραιότητες μεταξύ των τελεστών, άλλες περισσότερο και άλλες λιγότερο αναμενόμενες, οπότε η σειρά εφαρμογής των τελεστών είναι σαφής.
- Σε περίπτωση που θέλουμε να επιβάλουμε συγκεκριμένο τρόπο εφαρμογής των τελεστών, πρέπει να χρησιμοποιούμε παρενθέσεις, ακόμα και αν είμαστε βέβαιοι ότι σε κάποιες περιπτώσεις δεν είναι απαραίτητες λόγω των προτεραιοτήτων των τελεστών.
- Φυσικά, πρέπει να αποφεύγονται υπερβολές. Στην εντολή

  ```
  x = y + z*w;
  ```

  δεν χρειάζονται παρενθέσεις αν θέλουμε η παράσταση στο δεξιό μέλος να είναι η `y + (z*w)` (λόγω της αναμενόμενης μεγαλύτερης προτεραιότητας του `*` έναντι του `+`, που όντως ισχύει). Βέβαια, η παράσταση `(y+z) * w` είναι διαφορετική από την προηγούμενη.

- Κάθε εντολή αντικατάστασης μπορεί να θεωρηθεί και σαν παράσταση με τιμή αυτήν την τιμής της παράστασης που αναθέτουμε στη μεταβλητή. Έτσι, ο κορμός του προγράμματος `capitalize.c`

```
ch = getchar();
while (ch != EOF) {
  if (ch >= 'a' && ch <= 'z')
    ch = ch - ('a'-'A');
  putchar(ch);
  ch = getchar();
}
```

  θα μπορούσε να γραφεί πιο συμπυκνωμένα ως εξής:

```
while ((ch = getchar()) != EOF) {
  if (ch >= 'a' && ch <= 'z')
    ch = ch - ('a'-'A');
  putchar(ch);
}
```

  Προσέξτε, στη συμπυκνωμένη εκδοχή, τις παρενθέσεις γύρω από την εντολή αντικατάστασης `ch = getchar()`. Είναι απαραίτητες γιατί ο τελεστής `!=` έχει μεγαλύτερη προτεραιότητα από τον `=`. Αν δεν τις βάζαμε, τι θα γινόταν;

- Για εντολές αντικατάστασης, τελεστές και παραστάσεις στην C, υπάρχει εκτεταμένη ανάλυση στις παραγράφους §2.5 έως §2.12 του [KR], που συνοδεύεται από πάρα πολλά ενδιαφέροντα παραδείγματα.
- Επίσης, το Κεφάλαιο 1 από το [KR] (σελ. 19–58) είναι μία πολύ περιεκτική προπαρασκευαστική εισαγωγή στην C, που εμπλέκει πολλές έννοιες οι οποίες αναλύονται εκτενέστερα στη συνέχεια του βιβλίου. Είναι εξαιρετικά χρήσιμη η εισαγωγή αυτή, αν μη τι άλλο για να προσπαθήσει κανείς να καταλάβει τα προγράμματα που περιλαμβάνονται εκεί και, γιατί όχι, να δοκιμάσει να λύσει και τις ασκήσεις του.

Εύρεση Μ.Κ.Δ. και Ε.Κ.Π. τυχαίων αριθμών

```
/* File: gcdlcm.c */
#include <stdio.h>
#include <stdlib.h>      /* Header file for srand and rand functions */
#include <time.h>                    /* Header file for time function */
#define COMPUTATIONS 8    /* Pairs of numbers to compute GCD and LCM */
#define MAXNUMB 1000                                 /* Maximum number */

int main()
{ int i, m, n, large, small, gcd, rem;
  long curtime, lcm;
  curtime = time(NULL);  /* Current time (in seconds since 1/1/1970) */
  printf("Current time is %ld\n\n", curtime);
  srand((unsigned int) curtime); /* Seed for random number generator */
  for (i=0 ; i < COMPUTATIONS ; i++) {
    m = rand() % MAXNUMB + 1;    /* Select 1st number in [1,MAXNUMB] */
    n = rand() % MAXNUMB + 1;    /* Select 2nd number in [1,MAXNUMB] */
    if (m > n) {                           /* Which number is larger? */
      large = m;                                     /* Larger is m */
      small = n;
    }
    else {
      large = n;                                     /* Larger is n */
      small = m;
    }
    while (small) {                          /* While small is not 0 */
      rem = large % small;          /* Find remainder of large/small */
      large = small;                  /* New large is previous small */
      small = rem;                   /* New small is remainder found */
    }
    gcd = large;    /* GCD of m and n equals to final value of large */
    lcm = (m*n)/gcd;                  /* LCM(m,n) * GCD(m,n) = m * n */
    printf("GCD(%3d,%3d) = %2d  LCM(%3d,%3d) = %6ld\n",
           m, n, gcd, m, n, lcm);
  }
}
```

```
% gcc -o gcdlcm gcdlcm.c
% ./gcdlcm
Current time is 1130666309

GCD(855,405) = 45  LCM(855,405) =   7695
GCD(125,139) =  1  LCM(125,139) =  17375
GCD(996,467) =  1  LCM(996,467) = 465132
GCD(191,890) =  1  LCM(191,890) = 169990
GCD(548,612) =  4  LCM(548,612) =  83844
GCD(378,531) =  9  LCM(378,531) =  22302
GCD(837,127) =  1  LCM(837,127) = 106299
GCD(938,656) =  2  LCM(938,656) = 307664
%
```

- Για την εύρεση τυχαίων ζευγαριών θετικών ακεραίων, χρησιμοποιείται η συνάρτηση `rand`, η οποία κάθε φορά που καλείται επιστρέφει ένα (ψευδο)τυχαίο αριθμό. Η αρχικοποίηση της γεννήτριας (ψευδο)τυχαίων αριθμών γίνεται καλώντας τη συνάρτηση `srand`, δίνοντάς της σαν παράμετρο την τρέχουσα χρονική στιγμή, όπως αυτή επιστρέφεται από τη συνάρτηση `time` (η παράμετρος `NULL` που δίνουμε στην `time` θα συζητηθεί στο μέλλον).

- Για την εύρεση του Μέγιστου Κοινού Διαιρέτη (Μ.Κ.Δ.) δύο αριθμών, χρησιμοποιούμε τον αλγόριθμο του Ευκλείδη. Δηλαδή, βρίσκουμε το υπόλοιπο της διαίρεσης του μεγαλύτερου με τον μικρότερο από τους αριθμούς και αντικαθιστούμε τον μεγαλύτερο με τον μικρότερο, τον μικρότερο με το υπόλοιπο της διαίρεσης και επαναλαμβάνουμε τη διαδικασία μέχρι ο μικρότερος να γίνει ίσος με 0. Τότε ο ζητούμενος Μ.Κ.Δ. είναι ο μεγαλύτερος.

- Για την εύρεση του Ε.Κ.Π., χρησιμοποιούμε τη σχέση $\text{ΕΚΠ}(m,n) \cdot \text{ΜΚΔ}(m,n) = m \cdot n$.

Η ροή του ελέγχου στην C

- Όπως σε κάθε γλώσσα προγραμματισμού, έτσι και στην C, υπάρχουν εντολές που επηρεάζουν τη ροή του ελέγχου στα προγράμματα (`if`, `switch`, `while`, `for`, κλπ.).
- Στις εντολές αυτές είναι πολύ συχνή η χρήση ενός μπλοκ εντολών. Ένα μπλοκ εντολών αποτελείται από εντολές που είναι κλεισμένες μέσα σε άγκιστρα ({ και }). Αν σ΄ ένα μπλοκ εντολών περιέχεται μία μόνο εντολή, αυτή δεν είναι απαραίτητο να περικλείεται μέσα σε άγκιστρα.
- Είναι εξαιρετικά χρήσιμο να μελετηθεί όλο το Κεφάλαιο 3 από το [KR] (σελ. 85–100), στο οποίο περιγράφονται αναλυτικά όλες οι δομές ελέγχου που προσφέρει η C. Ακολουθεί μία πιο συνοπτική περιγραφή των δομών αυτών.

Εντολή `if`

- Η σύνταξη της εντολής `if` είναι:
  `if` (⟨παράσταση⟩)
  ⟨μπλοκ εντολών⟩$_1$
  `else`
  ⟨μπλοκ εντολών⟩$_2$
- Αν η ⟨παράσταση⟩ είναι αληθής (έχει τιμή διάφορη από το μηδέν), θα εκτελεσθεί το ⟨μπλοκ εντολών⟩$_1$, αλλιώς το ⟨μπλοκ εντολών⟩$_2$.
- Το τμήμα από το `else` και μετά είναι προαιρετικό. Αν δεν υπάρχει και η ⟨παράσταση⟩ είναι ψευδής, ο έλεγχος θα πάει στην επόμενη εντολή του προγράμματος, δηλαδή στην πρώτη εντολή μετά το `if`.
- Η προαιρετικότητα του `else` μπορεί να προκαλέσει προβλήματα στην αναγνωσιμότητα προγραμμάτων με εμφωλευμένες εντολές `if`, αν δεν υπάρχει σαφής ομαδοποίηση με άγκιστρα. Αν δεν υπάρχουν άγκιστρα που να επιβάλλουν συγκεκριμένη δομή, κάθε `else` αντιστοιχεί στην αμέσως προηγούμενη `if` που δεν έχει `else`.
- Τι ακριβώς σημαίνει το παρακάτω;

```
if (n > 0)
   if (a > b)
      z = a;
else
   z = b;
```

  Προσοχή! Ο μεταγλωττιστής δεν ασχολείται με τη στοίχιση.

- Αυτό:

```
if (n > 0) {
   if (a > b)
      z = a;
   else
      z = b;
}
```

- Αν θέλαμε το `else` να αντιστοιχεί στο πρώτο `if`, έπρεπε να το γράψουμε έτσι:

```
if (n > 0) {
   if (a > b)
      z = a;
}
else
   z = b;
```

- Μία συνηθισμένη χρήση της εντολής `if` είναι όταν θέλουμε να ελέγξουμε μία σειρά από επάλληλες συνθήκες, και ανάλογα με το ποια θα βρεθεί ότι ισχύει πρώτη, να εκτελέσουμε ένα μπλοκ εντολών. Αυτό γίνεται ως εξής:

`if (`$\langle$παράσταση$\rangle_1$`)`
    $\langle$μπλοκ εντολών$\rangle_1$
`else if (`$\langle$παράσταση$\rangle_2$`)`
    $\langle$μπλοκ εντολών$\rangle_2$
.....
`else if (`$\langle$παράσταση$\rangle_{n-1}$`)`
    $\langle$μπλοκ εντολών$\rangle_{n-1}$
`else` $\langle$μπλοκ εντολών$\rangle_n$

- Η προηγούμενη εντολή `if` θα μπορούσε να γραφεί σε μία εκδοχή με στοίχιση ως εξής:

  `if` (⟨παράσταση⟩$_1$)
  ⟨μπλοκ εντολών⟩$_1$
  `else`
  `if` (⟨παράσταση⟩$_2$)
  ⟨μπλοκ εντολών⟩$_2$
  .....
  `else`
  `if` (⟨παράσταση⟩$_{n-1}$)
  ⟨μπλοκ εντολών⟩$_{n-1}$
  `else`
  ⟨μπλοκ εντολών⟩$_n$

  Η συγκεκριμένη στοίχιση, όμως, μάλλον κάνει πιο δυσανάγνωστη την εντολή, οπότε είναι καλό να μην την ακολουθήσει κάποιος.

- Ένα παράδειγμα:

```
if (x > 0) {
   sign = 1;
   printf("Number is positive\n");
}
else if (x < 0) {
   sign = -1;
   printf("Number is negative\n");
}
else {
   sign = 0;
   printf("Number is zero\n");
}
```

Εντολή `switch`

- Η σύνταξη της εντολής `switch` είναι:
  `switch` (⟨παράσταση⟩) {
  `case` ⟨σταθερά⟩$_1$:
  ⟨εντολές⟩$_1$
  `case` ⟨σταθερά⟩$_2$:
  ⟨εντολές⟩$_2$
  .....
  `default:`
  ⟨εντολές⟩
  }
  Η `switch` είναι μία διακλαδωμένη εντολή απόφασης που λειτουργεί όπως περιγράφεται στη συνέχεια.

- Η ⟨παράσταση⟩ πρέπει να είναι ακέραια. Υπολογίζεται η τιμή της.

- Κάθε ⟨σταθερά⟩$_i$ πρέπει να είναι ακέραια σταθερά και δεν πρέπει να υπάρχουν δύο `case` με την ίδια σταθερά.

- Αν η τιμή που έχει η ⟨παράσταση⟩ ισούται με κάποια ⟨σταθερά⟩$_i$, τότε ο έλεγχος μεταφέρεται στις ⟨εντολές⟩$_i$. Αν όχι, ο έλεγχος μεταφέρεται στις ⟨εντολές⟩ (μετά το `default`).

- Οι ⟨εντολές⟩$_i$ (και ⟨εντολές⟩) δεν είναι απαραίτητο να συνιστούν μπλοκ εντολών, δηλαδή να είναι κλεισμένες μέσα σε { και }.

- Μετά την εκτέλεση των εντολών σε μία `case`, ο έλεγχος μεταφέρεται στις εντολές της επόμενης `case`, εκτός αν τελευταία εντολή της προηγούμενης `case` είναι η `break`. Δεν είναι καλή πρακτική οι εντολές μίας `case` να συνεχίζουν με αυτές της επόμενης, εκτός αν ισχύει το επόμενο.
- Είναι δυνατόν για περισσότερες τιμές της μίας για την ⟨παράσταση⟩ να θέλουμε να εκτελεσθεί η ίδια ομάδα εντολών. Σ′ αυτήν την περίπτωση, τοποθετούμε τα αντίστοιχα `case` συνεχόμενα και βάζουμε την ομάδα εντολών στο τελευταίο απ′ αυτά.
- Η περίπτωση `default` είναι προαιρετική. Αν δεν υπάρχει και δεν ταιριάζει καμία `case`, ο έλεγχος μεταφέρεται μετά την εντολή `switch`.
- Παράδειγμα:

```
switch (x) {
   case 1:  printf("one\n");
            break;
   case 2:
   case 3:  printf("two or three\n");
            break;
   case 4:  printf("four\n");
            break;
   default: printf("other\n");
            break;
}
```

  Τι θα γινόταν αν έλειπαν κάποιες `break`; Γιατί βάλαμε `break` και στην `default` περίπτωση;

Εντολές βρόχου `while`

- Μία πιθανή σύνταξη της εντολής `while` είναι:
  `while (`⟨παράσταση⟩`)`
  ⟨μπλοκ εντολών⟩

- Αρχικά υπολογίζεται η ⟨παράσταση⟩. Αν έχει τιμή διάφορη του μηδενός (είναι αληθής) εκτελείται το ⟨μπλοκ εντολών⟩. Υπολογίζεται πάλι η ⟨παράσταση⟩ και ενόσω είναι αληθής, θα γίνεται ο κύκλος εκτέλεσης του ⟨μπλοκ εντολών⟩, μέχρι να γίνει η ⟨παράσταση⟩ ψευδής (δηλαδή ίση με το μηδέν).

- Παράδειγμα:

```
f = 1;
while (--n)
   f = f*(n+1);
```

  Τι υπολογίζει αυτό το `while`;

- Επίσης, θυμηθείτε και την εντολή `while` στο `http://www.di.uoa.gr/~ip/cprogs/capitalize.c`.

- Άλλη πιθανή σύνταξη της εντολής `while` είναι:
  `do`
  ⟨μπλοκ εντολών⟩
  `while (`⟨παράσταση⟩`)`

- Αρχικά εκτελείται το ⟨μπλοκ εντολών⟩. Μετά, υπολογίζεται η ⟨παράσταση⟩. Αν έχει τιμή διάφορη του μηδενός (είναι αληθής) εκτελείται πάλι το ⟨μπλοκ εντολών⟩ και συνεχίζεται ο κύκλος εκτέλεσής του, μέχρι να γίνει η ⟨παράσταση⟩ ψευδής (δηλαδή ίση με το μηδέν).

- Για λόγους καλύτερης αναγνωσιμότητας του προγράμματος, ακόμα και αν το ⟨μπλοκ εντολών⟩ αποτελείται από μία μόνο εντολή, συνήθως τη βάζουμε μέσα σε { και }.

- Παράδειγμα:

```
f = 1;
do {
   f = f*n;
} while (--n);
```

  Τι υπολογίζει αυτό το `while`;

- Επίσης, θυμηθείτε και την εντολή `do`/`while` στο `http://www.di.uoa.gr/~ip/cprogs/picomp.c`.

- Η βασική διαφορά των δύο εκδοχών της εντολής `while` είναι ότι στην πρώτη περίπτωση το ⟨μπλοκ εντολών⟩ μπορεί και να μην εκτελεσθεί καμία φορά, αν η ⟨παράσταση⟩ είναι αρχικά ψευδής, ενώ στην εκδοχή `do`/`while`, το ⟨μπλοκ εντολών⟩ θα εκτελεσθεί τουλάχιστον μία φορά, αφού η ⟨παράσταση⟩ ελέγχεται στο τέλος.

Εντολή βρόχου `for`

- Η σύνταξη της εντολής `for` είναι:
  `for (`⟨παράσταση⟩$_1$ `;` ⟨παράσταση⟩$_2$ `;` ⟨παράσταση⟩$_3$`)`
  ⟨μπλοκ εντολών⟩

- Διαδικαστικά, ισοδυναμεί με τις εξής εντολές:
  ⟨παράσταση⟩$_1$`;`
  `while (`⟨παράσταση⟩$_2$`) {`
  ⟨μπλοκ εντολών⟩
  ⟨παράσταση⟩$_3$`;`
  }

- Δηλαδή, αρχικά υπολογίζεται η ⟨παράσταση⟩$_1$. Συνήθως, πρόκειται για την αρχικοποίηση ενός δείκτη που ελέγχει μία επαναληπτική διαδικασία. Στη συνέχεια, εκτελείται επαναλαμβανόμενα το ⟨μπλοκ εντολών⟩, που αποτελεί το σώμα της διαδικασίας, ενόσω η ⟨παράσταση⟩$_2$, που συνήθως ελέγχει την τιμή του δείκτη ελέγχου, είναι αληθής. Πριν τη διεξαγωγή νέας επανάληψης, εκτελείται και η ⟨παράσταση⟩$_3$, που συνήθως μεταβάλλει τον δείκτη ελέγχου.

- Παράδειγμα:

```
f = 1;
for (i=1 ; i <= n ; i++)
   f = f*i;
```

  Τι υπολογίζει αυτό το `for`;

- Επίσης, θυμηθείτε και τις εντολές `for` στα
  `http://www.di.uoa.gr/~ip/cprogs/easter.c` και
  `http://www.di.uoa.gr/~ip/cprogs/magic.c`.

- Είναι δυνατόν κάποια από τις $\langle$παράσταση$\rangle_{1,2\ ή\ 3}$ να μην υπάρχει, συνήθως η πρώτη ή/και η τρίτη. Αν δεν υπάρχει η δεύτερη, τότε δεν γίνεται έλεγχος για τερματισμό του βρόχου, οπότε πρέπει αυτό να γίνει βιαίως με κάποιο τρόπο μέσα από το σώμα της επανάληψης. Σε κάθε περίπτωση, όποια $\langle$παράσταση$\rangle_i$ και να λείπει, τα `;` υπάρχουν κανονικά.

- Η εντολή
  `for ( ; ; )`
  $\langle$μπλοκ εντολών$\rangle$
  είναι η κλασική υλοποίηση ενός ατέρμονος βρόχου.

- Επίσης, στην C υπάρχει και ο τελεστής `,` (κόμμα), με τον οποίο μπορούμε να κατασκευάσουμε μία σύνθετη παράσταση που θα υπολογισθεί από αριστερά προς τα δεξιά. Η τιμή της σύνθετης παράστασης ισούται με την τιμή της δεξιότερης από τις απλές που την συνιστούν.

- Μερικές φορές, θέλουμε να γράψουμε μία εντολή `for` που θα ελέγχεται από περισσότερους του ενός δείκτες. Αυτό γίνεται χρησιμοποιώντας τον τελεστή `,` στην $\langle$παράσταση$\rangle_1$ και στην $\langle$παράσταση$\rangle_3$. Παράδειγμα:

```
for (i=0, j=n ; i <= j ; i++, j--)
   printf("%d\n", i*j);
```

Εντολές `break` και `continue`

- Μία χρήση της εντολής `break` είναι αυτή που είδαμε στην εντολή `switch`, για τον τερματισμό της ακολουθίας εντολών που θα εκτελεσθούν όταν ταιριάξει κάποια συγκεκριμένη `case`.

- Επίσης, με την εντολή `break`, μπορούμε να τερματίσουμε βιαίως τις επαναλήψεις ενός βρόχου (`while`, `do/while` ή `for`), πριν ισχύσει η συνθήκη τερματισμού (ή όταν ο βρόχος είναι ατέρμων). Παράδειγμα:

```
while (!finished) {
   ......................
   if (bad_data)
      break;
   ......................
}
```

- Με την εντολή `continue`, ο έλεγχος στο εσωτερικό ενός βρόχου μεταφέρεται στο τέλος του, για να αρχίσει η επόμενη επανάληψη. Παράδειγμα:

```
for (i=0 ; i < n ; i++) {
   if (a[i] < 0)  /* Ignore negative elements */
      continue;
   ............  /* Process positive elements */
}
```

Εντολή goto και ετικέτες

- Κάθε γλώσσα προγραμματισμού, έτσι και η C, έχει μία εντολή για ρητή μεταφορά του ελέγχου σε κάποιο άλλο σημείο (ετικέτα) του προγράμματος. Στην C, η εντολή αυτή είναι η goto.

- Η goto μπορεί να χρησιμοποιηθεί σε πολύ εξειδικευμένες περιπτώσεις, για παράδειγμα όταν θέλουμε να γίνει βίαιη έξοδος από εμφωλευμένους βρόχους, όταν ο έλεγχος βρίσκεται σε κάποιο εσωτερικό βρόχο. Αυτό δεν μπορεί να γίνει με την εντολή break. Παράδειγμα:

```
for (i=0 ; i < n ; i++)
   for (j=0 ; j < m ; j++)
      if (a[i] == b[j])
         goto found;
..............   /* Didn't find common element */
found:
..............   /* Found a[i] == b[j] */
```

- Η εντολή goto βλάπτει σοβαρά την ιδέα του δομημένου προγραμματισμού, γιατί μπορεί να οδηγήσει σε προγράμματα δυσανάγνωστα και δύσκολα συντηρήσιμα, και γι΄ αυτό πρέπει να χρησιμοποιείται σπάνια, αν όχι καθόλου. [α΄]

[α΄] Στα πλαίσια του συγκεκριμένου μαθήματος, απλά ΑΠΑΓΟΡΕΥΕΤΑΙ η χρήση της.

Δομή ενός προγράμματος C – Συναρτήσεις

- Μία συνάρτηση C είναι ένα αυτόνομο, πακεταρισμένο τμήμα προγράμματος που φέρει σε πέρας μία διαδικασία η οποία έχει σαφείς προδιαγραφές εισόδου και εξόδου και συγκεκριμένο όνομα.
- Συνήθως, κωδικοποιούμε σε μία συνάρτηση έναν αλγόριθμο που πρόκειται να χρησιμοποιήσουμε στο πρόγραμμά μας αρκετές φορές ή εκτιμούμε ότι είναι γενικότερης χρησιμότητας και θα μπορούσε να φανεί χρήσιμος και σε προγράμματα που θα γράψουμε στο μέλλον.
- Σε γενικές γραμμές, είναι καλή πρακτική να γράφουμε προγράμματα C που αποτελούνται από πολλές και μικρές συναρτήσεις, παρά από λίγες και μεγάλες.
- Ακριβώς μία από τις συναρτήσεις ενός προγράμματος C πρέπει να έχει το όνομα `main`. Είναι η συνάρτηση από την οποία θα ξεκινήσει η εκτέλεση του προγράμματος.
- Ορισμός συνάρτησης:
  ⟨τύπος⟩ ⟨όνομα⟩ (⟨τυπικές παράμετροι⟩)
  {
  ⟨δηλώσεις και εντολές⟩
  }
- Για τον ορισμό κάθε συνάρτησης, δηλώνουμε το ⟨όνομα⟩ που έχει και, μέσα σε παρενθέσεις, τα ονόματα και τους τύπους που έχουν οι, λεγόμενες, ⟨τυπικές παράμετροι⟩, εφ΄ όσον υπάρχουν.

- Οι τυπικές παράμετροι μίας συνάρτησης είναι μεταβλητές, στις οποίες δίνονται συγκεκριμένες τιμές κατά την κλήση της συνάρτησης και οι οποίες χρησιμοποιούνται μέσα στο σώμα της συνάρτησης (⟨δηλώσεις και εντολές⟩) για να επιτευχθεί ο στόχος της συνάρτησης.
- Το σώμα μίας συνάρτησης περικλείεται μέσα σε { και }.
- Μία συνάρτηση μπορεί να επιστρέψει κάποια τιμή στο όνομά της, η οποία να χρησιμοποιηθεί από την συνάρτηση που την κάλεσε. Ο ⟨τύπος⟩ της τιμής που επιστρέφει μία συνάρτηση δίνεται μπροστά από το όνομά της, στον ορισμό της.
- Αν δεν ορίζεται τύπος επιστροφής για μία συνάρτηση, θεωρείται ότι επιστρέφει `int`.
- Αν δεν ενδιαφερόμαστε να επιστρέψει μία συνάρτηση κάποια τιμή, πρέπει να ορίζουμε ότι επιστρέφει `void`, τον "κενό" τύπο. Ακόμα και αν έχουμε ορίσει μία συνάρτηση να επιστρέφει κάποιο τύπο, όχι `void`, δεν είναι τελικά υποχρεωτικό να το κάνει, και αν το κάνει, δεν είναι υποχρεωτικό για την καλούσα συνάρτηση να χρησιμοποιήσει την επιστρεφόμενη τιμή.
- Η επιστροφή τιμής από συνάρτηση γίνεται με την εντολή `return` ⟨παράσταση⟩. Υπολογίζεται η τιμή που έχει η ⟨παράσταση⟩, επιστρέφεται και η συνάρτηση τερματίζει. Με μία απλή εντολή `return` τερματίζει χωρίς να επιστραφεί κάποια τιμή.

- Η συνάρτηση `main` μπορεί να ορίζεται
  1. είτε χωρίς τυπικές παραμέτρους
  2. είτε με δύο παραμέτρους συγκεκριμένων τύπων, μέσω των οποίων μπορεί να γνωρίζει η `main` αν το πρόγραμμα εκτελέσθηκε με ορίσματα στη γραμμή εντολής και, αν ναι, με ποια. Περισσότερα, γι΄ αυτό το θέμα, αργότερα.
- Η `main` επιστρέφει τιμή τύπου `int`, η οποία είναι διαθέσιμη στο περιβάλλον του λειτουργικού συστήματος από το οποίο εκτελέσθηκε το πρόγραμμα. Τελικά, ο απολύτως σωστός τρόπος για να ορίσουμε τη συνάρτηση `main` χωρίς τυπικές παραμέτρους, τον οποίο θα υιοθετήσουμε από εδώ και πέρα, είναι ο εξής (το `void` υποδηλώνει την έλλειψη τυπικών παραμέτρων):

  ```
  int main(void)
  ```

- Επίσης, πάντα θα πρέπει να επιστρέφουμε μία τιμή τερματισμού από τη `main`, με την εντολή `return` (τυπικά το 0, σε περίπτωση επιτυχούς τερματισμού του προγράμματος).
- Ο μεταγλωττιστής απαιτεί πριν την κλήση μίας συνάρτησης να γνωρίζει τους τύπους των τυπικών παραμέτρων της και τον τύπο της επιστρεφόμενης τιμής. Συνεπώς, ή πρέπει να προηγείται ο ορισμός μίας συνάρτησης από την κλήση της, ή, αν έπεται, να έχει προηγηθεί απλώς μία προαναγγελία της συνάρτησης (τύπος επιστρεφόμενης τιμής, όνομα και τύποι τυπικών παραμέτρων), το λεγόμενο πρωτότυπο της συνάρτησης.
- Περισσότερα για συναρτήσεις, στις παραγράφους §4.1 και §4.2 του [KR].

Συνάρτηση ύψωσης σε δύναμη

```
/* File: powers.c */
#include <stdio.h>
#define MAXM 4          /* Maximum base of powers to compute */
#define MAXN 6                            /* Maximum exponent */

int power(int, int);                    /* Prototype of power */

int main(void)
{ int m, n, p;
  for (m=2 ; m <= MAXM ; m++)            /* Start from base 2 */
    for (n=2 ; n <= MAXN ; n++) {    /* Start from exponent 2 */
       p = power(m,n);                 /* Call power function */
       printf("%d^%d = %d\n", m, n, p);
    }
  return 0;
}

int power(int base, int n)
{ int p;
  for (p=1 ; n > 0 ; n--)                 /* base^n equals to */
    p *= base;          /* base * base * ... * base (n times) */
  return p;                                  /* Return result */
}
```

```
% gcc -o powers powers.c
% ./powers
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
3^2 = 9
3^3 = 27
3^4 = 81
3^5 = 243
3^6 = 729
4^2 = 16
4^3 = 64
4^4 = 256
4^5 = 1024
4^6 = 4096
%
```

Συνάρτηση υπολογισμού παραγοντικού

```
/* File: factorial.c */
#include <stdio.h>
#define MAXN 12             /* Compute factorials up to MAXN */

int factorial(int n)                  /* No prototype needed */
{ int f;
  for (f=1 ; n > 0 ; n--)       /* Initialize factorial to 1 */
    f *= n;         /* Multiply into factorial n, n-1, ..., 1 */
  return f;                                  /* Return result */
}

int main(void)
{ int n;                          /* For all n from 1 to MAXN */
  for (n=1 ; n <= MAXN ; n++)           /* Call function and */
    printf("%2d! = %d\n", n, factorial(n));  /* print result */
  return 0;
}
```

```
% gcc -o factorial factorial.c
% ./factorial
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
%
```

Εμβέλεια και χρόνος ζωής μεταβλητών

- Οι μεταβλητές που ορίζονται μέσα σε μία συνάρτηση λέγονται αυτόματες ή τοπικές. Ο λόγος είναι ότι οι μεταβλητές αυτές έχουν περιορισμένη εμβέλεια και χρόνο ζωής. Ισχύουν μόνο μέσα στη συνάρτηση που έχουν ορισθεί και για όσο χρόνο εκτελείται η συνάρτηση.
- Εκτός από τις αυτόματες μεταβλητές, μπορούμε να ορίσουμε σ΄ ένα πρόγραμμα C και εξωτερικές ή καθολικές μεταβλητές. Οι μεταβλητές αυτές ορίζονται όπως και οι αυτόματες (με τον τύπο τους, το όνομά τους και με πιθανή αρχικοποίησή τους), μόνο που ορίζονται έξω από συναρτήσεις, στο ίδιο επίπεδο με αυτές.
- Οι εξωτερικές μεταβλητές έχουν το προτέρημα ότι είναι ορατές από κάθε συνάρτηση, μπορούν δηλαδή να χρησιμοποιηθούν στα σώματα των συναρτήσεων, και διατηρούνται καθ΄ όλη τη διάρκεια της εκτέλεσης του προγράμματος.
- Μέσω των εξωτερικών μεταβλητών, μπορεί να επιτευχθεί επικοινωνία μεταξύ συναρτήσεων χωρίς να χρειάζεται δεδομένα που κάποια συνάρτηση πρέπει να περάσει σε κάποια άλλη να το κάνει αυτό μέσω παραμέτρων.

- Το πλεονέκτημα που προσφέρουν οι εξωτερικές μεταβλητές μπορεί να μετατραπεί σε μειονέκτημα, αν δεν χρησιμοποιούνται με φειδώ. Προγράμματα που χρησιμοποιούν ευρέως εξωτερικές μεταβλητές μπορεί να είναι πολύ δυσανάγνωστα και δύσκολα συντηρήσιμα. Επίσης, συναρτήσεις που βασίζονται σε εξωτερικές μεταβλητές δεν είναι γενικής χρήσης.

- Αν ένα πρόγραμμα είναι διασπασμένο σε πολλά αρχεία και θέλουμε να έχουμε μία εξωτερική μεταβλητή που να είναι ορατή από τις συναρτήσεις σε όλα τα αρχεία, τότε πρέπει σε ακριβώς ένα αρχείο να δώσουμε τον ορισμό της μεταβλητής, για παράδειγμα

  ```
  int sp;
  ```

  ενώ σε καθένα από τα άλλα αρχεία πρέπει να κάνουμε μία δήλωση της μεταβλητής, προτάσσοντας τον προσδιοριστή `extern`, για παράδειγμα

  ```
  extern int sp;
  ```

  Ουσιαστικά, ο μεταγλωττιστής δεσμεύει μνήμη για μία εξωτερική μεταβλητή όταν δει τον ορισμό της (γι΄ αυτό πρέπει να υπάρχει αυτός ακριβώς μία φορά) και ενημερώνεται για τον τύπο μίας μεταβλητής όταν δει μία δήλωσή της με `extern` (θυμηθείτε ότι αν ένα πρόγραμμα είναι διασπασμένο σε πολλά αρχεία, τα αρχεία αυτά μεταγλωττίζονται ξεχωριστά).

- Αν θέλουμε η εμβέλεια μίας εξωτερικής μεταβλητής να είναι μόνο το αρχείο στο οποίο ορίζεται και να μην συγχέεται με κάποια εξωτερική μεταβλητή με το ίδιο όνομα, αλλά σε άλλο αρχείο, μπορούμε να προτάξουμε του ορισμού της τον προσδιοριστή `static`, για παράδειγμα

```
static double sum;
```

- Ο προσδιοριστής `static` μπορεί να εφαρμοσθεί και σε μεταβλητές που ορίζονται μέσα σε συνάρτηση, όμως με άλλη σημασία. Η εμβέλεια μίας τοπικής μεταβλητής που έχει ορισθεί σαν `static` μέσα σε συνάρτηση, είναι η συνάρτηση αυτή, αλλά η μεταβλητή δεν είναι αυτόματη. Διατηρείται, όπως και η τιμή της, και μετά τον τερματισμό της συνάρτησης. Σε νέα κλήση της συνάρτησης, η μεταβλητή έχει την τιμή που πήρε μετά την προηγούμενη κλήση.
- Μη αρχικοποιημένες εξωτερικές μεταβλητές θεωρούνται ότι έχουν τιμή 0.
- Μη αρχικοποιημένες αυτόματες μεταβλητές έχουν απροσδιόριστες τιμές.
- Οι τυπικές παράμετροι μίας συνάρτησης είναι αυτόματες μεταβλητές για τη συνάρτηση.

- Οι αυτόματες μεταβλητές σε μία συνάρτηση μπορούν να ορίζονται, εκτός από το επίπεδο του σώματος της συνάρτησης, και σε πιο εσωτερικά μπλοκ εντολών (για παράδειγμα στο σώμα μίας εντολής `while`). Αν υπάρχουν πολλαπλοί ορισμοί για το ίδιο όνομα μεταβλητής σε διάφορα μπλοκ, μέσα σε καθένα απ΄ αυτά ισχύει ο ορισμός που έγινε σ΄ αυτό, ή, αν δεν υπάρχει, αυτός που έγινε στο αμέσως πιο εξωτερικό του, ή, αν δεν υπάρχει, στο πιο εξωτερικό, κ.ο.κ.
- Αν υπάρχει σύγκρουση ονόματος μεταξύ μίας εξωτερικής μεταβλητής και μίας αυτόματης μέσα σ΄ ένα μπλοκ, ισχύει η αυτόματη.
- Παράδειγμα:

```
int count;
float y;

double myfun(int y)
{  int i;
   ..............
   while (......) {
      int i, j;
      ..........
      for (.......) {
         int count;
         ..........
      }
   }
}
```

Ποιες μεταβλητές ισχύουν πού;

- Όταν μεταγλωττίζεται ένα πρόγραμμα, δεσμεύεται ο απαιτούμενος χώρος για να φυλαχθούν όλες οι εξωτερικές μεταβλητές του προγράμματος (αλλά και οι τοπικές μεταβλητές που έχουν δηλωθεί σαν `static` μέσα σε συναρτήσεις). Ο χώρος αυτός είναι των στατικών (καθολικών) δεδομένων και είναι εκ των προτέρων γνωστός και δεσμευμένος μέσα στο εκτελέσιμο πρόγραμμα και αντιστοιχεί στο χώρο που θα καταληφθεί στη μνήμη όταν φορτωθεί το πρόγραμμα για να εκτελεσθεί.
- Οι αυτόματες μεταβλητές μίας συνάρτησης, ακριβώς επειδή έχουν περιορισμένο χρόνο ζωής, έχουν ένα διαφορετικό τρόπο φύλαξης, μέσω της, λεγόμενης, στοίβας.
- Η στοίβα για ένα πρόγραμμα είναι μία δομή τύπου LIFO (Last In First Out), δηλαδή μπορούμε να εισάγουμε διαδοχικά στοιχεία σ΄ αυτήν, αλλά αν θέλουμε να εξαγάγουμε κάποιο, θα εξαχθεί αυτό που εισήχθη πιο πρόσφατα.
- Όταν καλείται μία συνάρτηση, ο έλεγχος μεταφέρεται σ΄ αυτήν, αλλά πριν αρχίσει η εκτέλεσή της, δεσμεύεται χώρος στη στοίβα για τις αυτόματες μεταβλητές της (συμπεριλαμβανομένων και των τυπικών παραμέτρων της). Αν η συνάρτηση αυτή καλέσει στο σώμα της άλλη συνάρτηση, ακολουθείται η ίδια διαδικασία.

- Όταν τερματίζει μία συνάρτηση, επιστρέφει, αν χρειάζεται, κάποια τιμή στη συνάρτηση που την κάλεσε και ελευθερώνεται ο χώρος στη στοίβα που είχε κρατηθεί για το περιβάλλον (αυτόματες μεταβλητές) της συνάρτησης.
- Ο μηχανισμός φύλαξης των αυτόματων μεταβλητών των συναρτήσεων στη στοίβα επιτρέπει τη χρήση μίας πολύ ισχυρής προγραμματιστικής δυνατότητας, της *αναδρομής* (recursion).
- Μία συνάρτηση είναι αναδρομική όταν καλεί τον εαυτό της. Λόγω της στοίβας, δεν υπάρχει κίνδυνος σύγχυσης με τις μεταβλητές της συνάρτησης και τις τιμές τους, αφού σε κάθε κλήση δημιουργείται νέο περιβάλλον στη στοίβα για τις μεταβλητές της συγκεκριμένης ενεργοποίησης.
- Φυσικά, πρέπει η αναδρομική κλήση μίας συνάρτησης να γίνεται υπό συνθήκη, γιατί αλλιώς η εκτέλεση του προγράμματος δεν θα μπορούσε να γίνει με πεπερασμένο τρόπο.
- Για εξωτερικές και αυτόματες μεταβλητές, αλλά και για την αναδρομή, δείτε τις παραγράφους §4.3, §4.4 και §4.6 έως §4.10 του [KR].

Υπολογισμός παραγοντικού με αναδρομή

```
/* File: recfact.c */
#include <stdio.h>
#define MAXN 12             /* Compute factorials up to MAXN */

int recfact(int);                    /* Prototype of recfact */

int main(void)
{ int n;                          /* For all n from 0 to MAXN */
  for (n=0 ; n <= MAXN ; n++)           /* Call function and */
    printf("%2d! = %d\n", n, recfact(n));    /* print result */
  return 0;
}

int recfact(int n)
{ if (!n)                                        /* If n == 0 */
    return 1;                                       /* 0! = 1 */
  else
    return n*recfact(n-1);                /* n! = n * (n-1)! */
}
```

```
% gcc -o recfact recfact.c
% ./recfact
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
%
```

Μετατροπές μεταξύ δεκαδικών και δυαδικών αριθμών

```
/* File: convdecbin.c */
#include <stdio.h>
#define ERROR -1       /* Return value for illegal character */

int getinteger(int base)
{ char ch; /* No need to declare ch as int - no EOF handling */
  int val = 0;                    /* Initialize return value */
  while ((ch = getchar()) != '\n')    /* Read up to new line */
    if (ch >= '0' && ch <= '0'+base-1)   /* Legal character? */
      val = base*val + (ch-'0');      /* Update return value */
    else
      return ERROR;                /* Illegal character read */
  return val; /* Everything OK - Return value of number read */
}

int main(void)
{ int n, digs[32], ind = 0;
  printf("Please, give a binary number:  ");
  n = getinteger(2);                   /* Read binary number */
  if (n == ERROR) {
    printf("Sorry, illegal character\n");  return 1; }
  else                        /* Print decimal value of number */
    printf("Decimal equivalent is:         %d\n", n);
  printf("Please, give a decimal number: ");
  n = getinteger(10);                 /* Read decimal number */
  if (n == ERROR) {
    printf("Sorry, illegal character\n");  return 1; }
  else {                    /* Convert number to binary digits */
    while (n) {             /* Repeat until number equals to 0 */
      digs[ind++] = n%2;         /* New digit is number mod 2 */
      n = n/2;                      /* New number is number/2 */
    }
    printf("Binary equivalent is:          ");
    while (ind--)             /* Print digits in reverse order */
      printf("%d", digs[ind]);
    printf("\n");
  }
  return 0;
}
```

```
% gcc -o convdecbin convdecbin.c
% ./convdecbin
Please, give a binary number:  10001001011111010001001001
Decimal equivalent is:         18019401
Please, give a decimal number: 18019401
Binary equivalent is:          10001001011111010001001001
% ./convdecbin
Please, give a binary number:  10011101
Decimal equivalent is:         157
Please, give a decimal number: 65535
Binary equivalent is:          1111111111111111
%
```

- Αν θεωρούσαμε ότι η συνάρτηση `getinteger` είναι γενικής χρησιμότητας, τι ακριβώς θα έπρεπε να βάλουμε σ΄ ένα πηγαίο αρχείο `getinteger.c` και τι σ΄ ένα αρχείο επικεφαλίδας `getinteger.h`, ώστε να χρησιμοποιούμε τη συνάρτηση αυτή και σε άλλα προγράμματα;

- Και ένα πρόβλημα:
  Η ακολουθία Fibonacci ορίζεται ως εξής:

$$a_0 = 0$$

$$a_1 = 1$$

$$a_n = a_{n-1} + a_{n-2}, \quad \forall\ n \geq 2$$

  Γράψτε πρόγραμμα C που να υπολογίζει, για δεδομένο k, τον $(k+1)$-οστό όρο a_k της ακολουθίας Fibonacci. Υλοποιήστε δύο διαφορετικές εκδοχές του προγράμματος, μία χωρίς αναδρομή και μία με αναδρομή. Έχετε να σχολιάσετε κάτι;

Δείκτες

- Ένας δείκτης στην C είναι μία μεταβλητή στην οποία μπορούμε να καταχωρήσουμε κάποια διεύθυνση μνήμης. Στη διεύθυνση αυτή μπορούμε να φυλάξουμε κάποια τιμή συγκεκριμένου τύπου.

- Η απλή εκδοχή μίας δήλωσης μεταβλητής τύπου δείκτη είναι της μορφής:
  ⟨τύπος⟩ *⟨μεταβλητή⟩
  Με τη δήλωση αυτή ορίζουμε τη ⟨μεταβλητή⟩ να είναι δείκτης σε δεδομένα που ο τύπος τους είναι ⟨τύπος⟩.

- Μπορούμε όμως να ορίσουμε μία ⟨μεταβλητή⟩ να είναι δείκτης σε δεδομένα τύπου δείκτη που δείχνουν σε δεδομένα που ο τύπος τους είναι ⟨τύπος⟩. Αυτό μπορεί να γίνει με μία δήλωση της μορφής:
  ⟨τύπος⟩ **⟨μεταβλητή⟩

- Δεν υπάρχει αμφιβολία ότι η τακτική αυτή γενικεύεται και σε πιο πολύπλοκες δηλώσεις (δείκτης σε δείκτη σε δείκτη κλπ.), αλλά οι πραγματικά χρήσιμοι δείκτες, στη μεγάλη πλειοψηφία των προγραμμάτων που γράφουμε, είναι αυτοί που ορίζουμε με τις παραπάνω δηλώσεις.

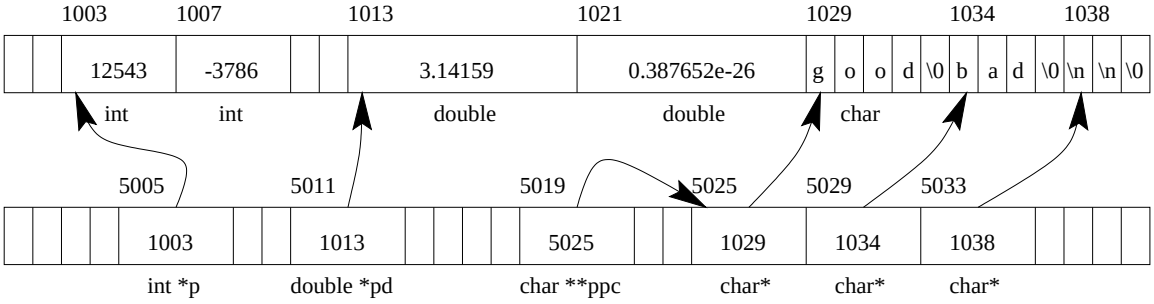
- Περί δεικτών, αλλά και πινάκων, που θα συζητήσουμε στη συνέχεια, το Κεφάλαιο 5 από το [KR] (σελ. 135–178) είναι μία ανεξάντλητη πηγή πληροφοριών. [α΄]

[α΄] Εκτός, ίσως, από τα θέματα της δυναμικής δέσμευσης μνήμης, που υπάρχει μία σχετική δυστοκία.

- Παράδειγμα:

```
int *p;
double *pd;
char **ppc;
```

Οι μεταβλητές p και pd ορίστηκαν σαν δείκτες σε ακεραίους και πραγματικούς διπλής ακρίβειας, αντίστοιχα.
Η μεταβλητή ppc είναι δείκτης σε θέση μνήμης που μπορούμε να φυλάξουμε δείκτη σε χαρακτήρες.
Δείτε και το σχήμα:



- Το σύμβολο * εφαρμοσμένο σε μεταβλητές τύπου δείκτη χρησιμοποιείται και στις εντολές του προγράμματος, ως τελεστής έμμεσης αναφοράς. Όταν έχουμε ορίσει μία ⟨μεταβλητή⟩ τύπου δείκτη, η έκφραση *⟨μεταβλητή⟩ παριστάνει το περιεχόμενο της θέσης μνήμης στην οποία δείχνει η ⟨μεταβλητή⟩.

- Ο αντίστροφος του τελεστή * είναι ο &. Η έκφραση &⟨μεταβλητή⟩ παριστάνει τη διεύθυνση που φυλάσσεται η τιμή για τη ⟨μεταβλητή⟩, ό,τι τύπου και να είναι αυτή. Δηλαδή, οι τελεστές * και & διαβάζονται ως εξής:
  &p = *η διεύθυνση μνήμης που είναι αποθηκευμένο το* p
  *p = *το περιεχόμενο της θέσης μνήμης που δείχνει το* p

- Παράδειγμα:

```
int x = 5, y, *px, *py, *pz;
char c = 'F', *pc, d;
px = &x;
py = &y;
pc = &c;
pz = py;
*pz = (*px)++;
d = --(*pc);
pc = &d;
(*pc)--;
```

  Ποιες οι τιμές των x, y, c, d μετά από αυτές τις εντολές; [α΄]

- Θα μπορούσαμε να είχαμε γράψει --*pc αντί για --(*pc); [β΄]

- Επίσης, γράψαμε (*px)++. Οι παρενθέσεις χρειάζονται στην έκφραση αυτή, αν θέλουμε ο τελεστής μοναδιαίας αύξησης ++ να εφαρμοσθεί επάνω στο περιεχόμενο της θέσης μνήμης που δείχνει ο δείκτης px. Αν είχαμε γράψει *px++, λόγω της υψηλότερης προτεραιότητας του τελεστή ++ (αλλά και του --) έναντι του τελεστή *, αυτό θα σήμαινε το *(px++).

[α΄] 6, 5, 'E' και 'D', αντίστοιχα

[β΄] Ναι, γιατί όχι;

- Τι σημαίνει όμως το `*(px++)` (ή το ισοδύναμό του `*px++`); [α΄]
- Όμως, ποιο είναι το νόημα να αυξήσουμε ή να μειώσουμε ένα δείκτη κατά 1, δηλαδή να κάνουμε τον δείκτη να δείχνει στην επόμενη ή προηγούμενη θέση μνήμης από αυτή που έδειχνε; Έχει νόημα αν αναφερόμαστε σε συνεχόμενες θέσεις μνήμης που, με κάποιο τρόπο, έχουμε δεσμεύσει και χρησιμοποιούμε.
- Εκτός από την αύξηση ή μείωση ενός δείκτη κατά 1 (μέσω των τελεστών μοναδιαίας αύξησης και μείωσης `++` και `--`), μπορούμε να προσθέσουμε σε ένα δείκτη ή να αφαιρέσουμε από αυτόν μία ακέραια σταθερά.
- Άλλες πράξεις μεταξύ δεικτών δεν επιτρέπονται, εκτός από την αφαίρεση δεικτών που δείχνουν σε στοιχεία ενός μπλοκ από δεδομένα του ίδιου τύπου, που έχουν δεσμευθεί, με κάποιο τρόπο, για ενιαία διαχείριση. Επίσης, επιτρέπονται και συγκρίσεις μεταξύ τέτοιων δεικτών.

[α΄] Όχι, ΔΕΝ σημαίνει ότι πρώτα θα αυξηθεί ο δείκτης `px` και μετά θα πάρουμε το περιεχόμενο της θέσης μνήμης που δείχνει η νέα, αυξημένη, τιμή του δείκτη. Οι παρενθέσεις δεν δείχνουν τη σειρά που θα γίνουν οι υπολογισμοί, αλλά το πού εφαρμόζονται οι τελεστές. Στο προκείμενο παράδειγμα, ο τελεστής `++` είναι μεταθεματικός, άρα πρώτα θα πάρουμε το περιεχόμενο της θέσης μνήμης που δείχνει ο δείκτης `px` και μετά θα αυξηθεί ο δείκτης. Αν θέλαμε πρώτα να αυξήσουμε τον δείκτη και μετά να κάνουμε την αναφορά, έπρεπε να γράψουμε `*(++px)`.

- Αν οι `pi`, `pc` και `pd` είναι δείκτες (σε `int`, `char` και `double`, αντίστοιχα), οι παρακάτω εντολές είναι απολύτως νόμιμες:

```
pi = pi+3;
pc -= 4;
pd = pd-2;
```

  Η πρόσθεση σε (αφαίρεση από) ένα δείκτη `p` μίας σταθεράς `n`, μεταβάλλει τον δείκτη ώστε να δείξει `n` θέσεις μνήμης (όχι `n` bytes), μεγέθους όσο αυτό του τύπου δεδομένων που δείχνει ο δείκτης, πιο μετά (πριν). Αν οι `int`, `char` και `double` είναι των 4, 1 και 8 bytes, αντίστοιχα, πόσο θα μεταβληθούν οι δείκτες `pi`, `pc` και `pd`; [α΄]

- Μία πολύ συνηθισμένη χρήση των δεικτών είναι στην περίπτωση που θέλουμε κάποια συνάρτηση να επιστρέψει κάποια τιμή (ή και περισσότερες), όχι σαν επιστρεφόμενη τιμή στο όνομά της, αλλά επιδρώντας στις μεταβλητές της συνάρτησης που την κάλεσε. Τότε, η καλούσα συνάρτηση πρέπει να περάσει στην καλουμένη τη διεύθυνση μίας μεταβλητής της, στην οποία η καλουμένη θα γράψει την τιμή που πρέπει να επιστρέψει.

- Θεωρήστε το εξής πρόβλημα: Θέλουμε να γράψουμε μία συνάρτηση `swap`, η οποία να ανταλλάσσει τα περιεχόμενα δύο ακέραιων μεταβλητών.

[α΄] `12`, `-4` και `-16`, αντίστοιχα

- Έστω ότι για να ανταλλάξουμε τις τιμές των ακέραιων μεταβλητών a και b προτιθέμεθα να καλέσουμε τη συνάρτηση swap(a,b), την οποία έχουμε ορίσει ως εξής:

```
void swap(int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp; }
```

  Γιατί είναι λάθος αυτό; [α΄]

- Πώς θα γράφαμε τη σωστή swap; Έτσι:

```
void swap(int *px, int *py)
{ int temp;
  temp = *px;
  *px = *py;
  *py = temp; }
```

  Και πώς θα την καλούσαμε για να ανταλλάξουμε τις τιμές των μεταβλητών a και b; [β΄]

- Και γιατί χρειαζόμαστε τη μεταβλητή temp; [γ΄]

[α΄] Έστω ότι πριν καλέσουμε την swap(a,b), οι τιμές των μεταβλητών a και b είναι 5 και 8, αντίστοιχα. Τότε, ουσιαστικά καλούμε swap(5,8), δηλαδή οι τυπικές παράμετροι x και y της swap, που είναι τοπικές/αυτόματες μεταβλητές για τη συνάρτηση, παίρνουν τις τιμές 5 και 8, αντίστοιχα, ανταλλάσσονται μέσα στην swap οι τιμές των x και y, αλλά οι μεταβλητές a και b της καλούσας συνάρτησης δεν επηρεάζονται καθόλου από την αλλαγή αυτή.

[β΄] swap(&a,&b)

[γ΄] Για τον ίδιο λόγο που χρειαζόμαστε ένα τρίτο ποτήρι αν θέλουμε να ανταλλάξουμε τα περιεχόμενα δύο άλλων ποτηριών, εκτός κι αν είμαστε ταχυδακτυλουργοί (:-).

Περί ανάγνωσης ακεραίων (και όχι μόνο)

- Μέχρι στιγμής, δεν γνωρίζαμε κάποιο εύχρηστο τρόπο να διαβάσουμε έναν ακέραιο μέσα από ένα πρόγραμμα C. Τώρα, με τη χρήση δεικτών, αυτό είναι εφικτό.
- Η πρότυπη βιβλιοθήκη εισόδου/εξόδου της C παρέχει και τη συνάρτηση `scanf`, που είναι η "αδελφή" συνάρτηση της `printf`, και με την οποία μπορούμε να διαβάσουμε από την είσοδο ενός προγράμματος δεδομένα προς επεξεργασία.
- Η `scanf` συντάσσεται με αντίστοιχο τρόπο αυτού της `printf`, δηλαδή στην πρώτη παράμετρο δίνουμε μία συμβολοσειρά που περιγράφει τι τύπων δεδομένα σκοπεύουμε να διαβάσουμε και με ποιον τρόπο. Οι επόμενες παράμετροι αναφέρονται στο πού θα φυλαχθούν οι τιμές που διαβάστηκαν.
- Μόνο ΠΡΟΣΟΧΗ! Επειδή η συνάρτηση `scanf` πρόκειται να επιστρέψει στη συνάρτηση που την κάλεσε κάποιες τιμές, στις αντίστοιχες παραμέτρους ΔΕΝ βάζουμε τις μεταβλητές στις οποίες θέλουμε να φυλαχθούν οι τιμές αυτές, αλλά δείκτες στις μεταβλητές αυτές. Είναι ο ίδιος ακριβώς λόγος για τον οποίο τη συνάρτηση `swap` την καλούμε σαν `swap(&a,&b)`, όπου `a` και `b` είναι ακέραιες μεταβλητές.

- Παράδειγμα:

```
int id, rank, *prank;
float salary;
prank = &rank;
printf("Please give id and rank: ");
scanf("%d %d", &id, prank);
printf("Please give salary: ");
scanf("%f", &salary);
```

- Η ανάγνωση τιμών για την ακέραια μεταβλητή `id` και τη μεταβλητή κινητής υποδιαστολής `salary` γίνεται περνώντας στις συναρτήσεις `scanf` δείκτες στις μεταβλητές αυτές. Όμως στην πρώτη `scanf` περάσαμε την ίδια τη μεταβλητή `prank`, αφού αυτή έχει ήδη ορισθεί σαν δείκτης σε ακέραιο. Βέβαια, στη συνέχεια, για να αναφερθούμε στην τιμή που διαβάσαμε, αυτό πρέπει να γίνει μέσω του τελεστή έμμεσης αναφοράς, δηλαδή `*prank`, ή μέσω της μεταβλητής `rank`.

- Οι προδιαγραφές `%d` και `%f` στις `scanf` υποδεικνύουν ότι οι τιμές που θα διαβαστούν είναι ακέραιες και κινητής υποδιαστολής, αντίστοιχα.

- Υπάρχουν και άλλες προδιαγραφές που μπορεί να χρησιμοποιηθούν για ανάγνωση τιμών από την `scanf`, καθώς και κάποιες ενδιαφέρουσες δυνατότητες που παρέχονται από τη συνάρτηση. Για περισσότερα, "`man scanf`" ή όταν θα αναφερθούμε αναλυτικότερα στις συναρτήσεις της πρότυπης βιβλιοθήκης εισόδου/εξόδου της C.

Πίνακες

- Όταν θέλουμε στην C να χειριστούμε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο, ορίζουμε ένα πίνακα συγκεκριμένου μεγέθους για τη φύλαξη των δεδομένων αυτών.

- Με τη δήλωση

  ```
  int myarray[20];
  ```

  ορίζουμε ένα μονοδιάστατο πίνακα ακεραίων με όνομα `myarray` στον οποίο μπορούν να φυλαχθούν το πολύ 20 ακέραιες τιμές. Το 20 είναι η διάσταση του πίνακα.

- Ένας πίνακας αποθηκεύεται στη μνήμη σε συνεχόμενες πάντα θέσεις.

- Μέσα στο πρόγραμμά μας, μπορούμε να αναφερόμαστε στα στοιχεία του πίνακα `myarray`, που έχει οριστεί με διάσταση 20, σαν `myarray[0]`, `myarray[1]`, ..., `myarray[19]`. Γενικά, με το `myarray[i]` αναφερόμαστε στο στοιχείο του πίνακα `myarray` με δείκτη[α΄] `i`, όπου το `i` μπορεί να είναι κάποια παράσταση, όχι μόνο σταθερά ή μεταβλητή, η οποία πρώτα θα αποτιμηθεί και μετά θα γίνει η προσπέλαση του στοιχείου `myarray[i]`.

[α΄] Προσοχή στην ορολογία! Τον όρο "δείκτης" εδώ, τον χρησιμοποιούμε σαν ελληνική απόδοση του Αγγλικού "index". Πρόκειται για δείκτη πίνακα. Οι δείκτες όμως που είδαμε στην προηγούμενη ενότητα είναι δείκτες διευθύνσεων και αντιστοιχούν στον Αγγλικό όρο "pointer". Για τη συνέχεια, δεν θα κάνουμε ιδιαίτερη αναφορά, όταν χρησιμοποιούμε τον όρο "δείκτης", σε ποια εκδοχή αναφερόμαστε. Θα προκύπτει αυτό εύκολα από τα συμφραζόμενα.

- Αν η διάσταση ενός πίνακα είναι `N`, οι δείκτες των στοιχείων του κυμαίνονται από `0` έως `N-1`.
- Προσοχή στο κλασικό λάθος που γίνεται στον προγραμματισμό με τη διαχείριση πινάκων! Απόπειρα πρόσβασης εκτός των ορίων ενός πίνακα δεν είναι δυνατόν να διαγνωσθεί από τον μεταγλωττιστή, με συνέπεια να προκύψει σοβαρό πρόβλημα κατά την εκτέλεση του προγράμματος.
- Είναι ευθύνη του προγραμματιστή να εξασφαλίσει ότι όταν στο πρόγραμμά του κάνει πρόσβαση στο στοιχείο `x[i]`, όπου ο πίνακας `x` έχει ορισθεί με διάσταση, έστω, `100`, τότε το `i` δεν θα έχει τιμή μικρότερη από το `0` ή μεγαλύτερη από το `99`, όταν γίνεται η αναφορά στο `x[i]`. Αλλιώς ...
  Στο Unix, για παράδειγμα, θα πάρουμε ένα μεγαλοπρεπές "`Segmentation fault`" ή κανένα "`Bus error`" κατά την εκτέλεση του προγράμματος. Γενικώς, αυτά τα μηνύματα μας υποδεικνύουν ότι έχουμε κάνει στο πρόγραμμά μας κάποιο λάθος διαχείρισης μνήμης.[α΄] Σε περιβάλλοντα Microsoft Windows, το πιο πιθανό αποτέλεσμα όταν υπάρχει λάθος διαχείρισης μνήμης είναι ο βίαιος τερματισμός του προγράμματος.

[α΄] Ευτυχώς, δηλαδή, γιατί έτσι μαθαίνουμε ότι το πρόγραμμά μας έχει σφάλματα, οπότε μπορούμε να μπούμε στη διαδικασία να τα διορθώσουμε. Όμως, αυτό δεν συμβαίνει πάντοτε. Υπάρχουν περιπτώσεις που να έχουμε κάνει λάθος διαχείρισης μνήμης, να μην πάρουμε κάποια ένδειξη γι΄ αυτό και να νομίζουμε ότι το πρόγραμμά μας δουλεύει σωστά, ενώ αυτό θα κάνει άλλα αντ΄ άλλων.

- Υπάρχει πολύ στενή σχέση μεταξύ των πινάκων και των δεικτών διευθύνσεων στην C. Ουσιαστικά, το όνομα ενός πίνακα είναι ένας σταθερός δείκτης στο πρώτο στοιχείο του πίνακα.

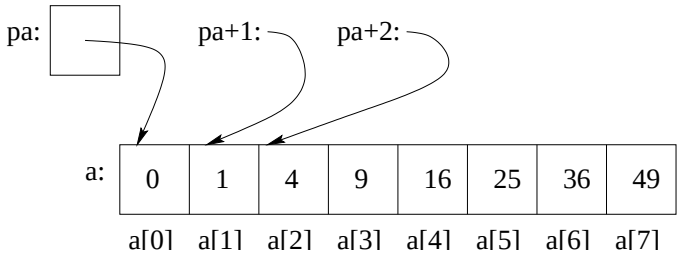
- Για παράδειγμα, αν έχουμε ορίσει έναν πίνακα `a`, τότε η έκφραση `*(a+i)` είναι ισοδύναμη με την `a[i]`. Ομοίως, και οι εκφράσεις `a+i` και `&a[i]` είναι ισοδύναμες.

- Αν σ΄ ένα δείκτη `pa`, που δείχνει σε στοιχεία ίδιου τύπου με τον τύπο των στοιχείων ενός πίνακα `a`, αναθέσουμε σαν τιμή τη διεύθυνση του πρώτου, ή κάποιου άλλου, στοιχείου του πίνακα, οι εκφράσεις `pa+`$\langle$N$\rangle$, όπου $\langle$N$\rangle$ είναι ακέραια παράσταση, είναι νόμιμες ως διευθύνσεις για προσπέλαση των στοιχείων του πίνακα, αρκεί να έχουμε εξασφαλίσει ότι δείχνουν μέσα στα όρια του πίνακα.

- Επίσης, μπορούμε τον δείκτη `pa` να τον αυξάνουμε ή να τον μειώνουμε, και, γενικά, να τον τροποποιούμε, κατά βούληση (π.χ. `pa++`, `--pa`, κλπ.), πάντα υπό την προϋπόθεση ότι μέσω του τροποποιημένου δείκτη θα προσπελαύνουμε στοιχεία του πίνακα εντός των ορίων του.

- Παράδειγμα:

```
int i, a[8], *pa;
for (i=0 ; i<8 ; i++)
  a[i] = i*i;
pa = &a[0];
a[6] = *(a+4);
*(pa+3) = a[5];
a[0] = *((pa++)+2);
*((++pa)+5) = a[1];
*(&a[5]-1) = *(--pa);
```



Ποια θα είναι τα περιεχόμενα του πίνακα a μετά την εκτέλεση των παραπάνω εντολών; [α΄]

- Προσέξτε ότι ενώ στο προηγούμενο παράδειγμα, τροποποιούσαμε τον δείκτη pa, αυτό δεν μπορεί να γίνει για το όνομα του πίνακα, παρότι είναι επιτρεπτό να χρησιμοποιηθεί σαν δείκτης (π.χ. *(a+4)). Δηλαδή, απαγορεύεται να γράψουμε a++.

[α΄] 4, 1, 4, 25, 1, 25, 16, 1

- Μερικές φορές, θέλουμε να ορίσουμε μία συνάρτηση, η οποία να επεξεργάζεται τα στοιχεία ενός πίνακα. Φυσικά, μία λύση είναι να έχει δηλωθεί ο πίνακας εξωτερικός, οπότε μπορεί να γίνει η επικοινωνία με την καλούσα συνάρτηση πολύ απλά.
- Είναι πιθανό, όμως, για διάφορους λόγους, κυρίως για να είναι η συνάρτησή μας γενικής χρήσης, να θέλουμε να περάσουμε τον πίνακα στη συνάρτηση σαν παράμετρο. Στην περίπτωση αυτή, απλώς περνάμε ένα δείκτη στο πρώτο στοιχείο του πίνακα και, μέσω αυτού, η συνάρτηση μπορεί να προσπελάσει οποιοδήποτε στοιχείο του.
- Παράδειγμα:

```
#define N 50

int main(void)
{  int x[N], *myp;
   ..............
   myfun(&x[0], N);
   ..............
}

void myfun(int *px, int n)
{  int y;
   ..............
   y = *(px+2);
   ..............
}
```

- Στο προηγούμενο παράδειγμα, η κλήση της συνάρτησης `myfun` από την `main` θα μπορούσε να είχε γίνει και σαν `myfun(x, N)`, δηλαδή να περάσουμε το όνομα του πίνακα, αφού, όπως γνωρίζουμε, το όνομα αυτό είναι ουσιαστικά ένας δείκτης στο πρώτο στοιχείο του πίνακα.

- Αν είχαμε αναθέσει στον δείκτη `myp` τη διεύθυνση του πρώτου στοιχείου του πίνακα (`myp = &x[0]` ή, ισοδύναμα, `myp = x`), θα μπορούσαμε να καλέσουμε τη συνάρτηση και σαν `myfun(myp, N)`.

- Αν θέλουμε να περάσουμε στη συνάρτηση έναν υποπίνακα του αρχικού πίνακα, από ένα στοιχείο και μετά, θα μπορούσαμε να την καλέσουμε με παράμετρο τη διεύθυνση του στοιχείου αυτού. Για παράδειγμα, αν έχουμε κάνει πρώτα την ανάθεση `myp = &x[2]` με την κλήση `myfun(myp, N-2)` (ή απ΄ ευθείας `myfun(&x[2], N-2)`), η συνάρτηση `myfun` θα "δει" έναν πίνακα με πρώτο στοιχείο το τρίτο του αρχικού.

- Επίσης, στον ορισμό της συνάρτησης, θα μπορούσαμε για τυπική παράμετρο να βάλουμε το `int px[]`, αντί για το `int *px`. Είναι απολύτως ισοδύναμα.

Ιστόγραμμα συχνοτήτων γραμμάτων στην είσοδο

```
/* File: histogram.c */
#include <stdio.h>
#define YAXISLEN 12                                    /* Y-axis length */

int main(void)
{ int i, j, ch, total;
  int letfr[26];         /* Letter occurrences and frequencies array */
  for (i=0 ; i < 26 ; i++)
    letfr[i] = 0;                                  /* Initialize array */
  total = 0;       /* Initialize counter of total letter occurrences */
  while ((ch = getchar()) != EOF) {/* Well-known read-character loop */
    if (ch >= 'A' && ch <= 'Z') {
      letfr[ch-'A']++;                     /* Found upper case letter */
      total++;
    }
    if (ch >= 'a' && ch <= 'z') {
      letfr[ch-'a']++;                     /* Found lower case letter */
      total++;
    }
  }
  printf("  |");  /* Start histogram printing - first Y-axis segment */
  for (i=0 ; i < 26 ; i++) {           /* Convert letter occurrences */
                        /* to frequencies rounded to nearest integer */
    letfr[i] = (int) ((100.0*letfr[i])/total+0.5);
    printf("%s", (letfr[i] > YAXISLEN) ? "^^" : "  ");    /* If i-th */
               /* letter frequency exceeds Y-axis length, print "^^" */
  }
  printf("\n");
  for (j=YAXISLEN ; j > 0 ; j--) {/* Print line at j-value of Y-axis */
    printf("%2d|", j);   /* Print frequency label and Y-axis segment */
    for (i=0 ; i < 26 ; i++)
      printf("%s", (letfr[i] >= j) ? "xx" : "  "); /* If i-th letter */
              /* frequency is greater than or equal to j, print "xx" */
    printf("\n");
  }                                 /* Print X-axis and letter labels */
  printf("  +--------------------------------------------------\n");
  printf("   AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz\n");
  return 0;
}
```

```
% gcc -o histogram histogram.c
% ./histogram < histogram.c
   |
12|
11|        xx                            xx
10|        xx      xx                    xx
 9|        xx      xx                xx  xx
 8|        xx      xx        xx      xx  xx
 7|        xx      xx        xx      xx  xx
 6|xx      xx      xx    xx  xx      xx  xx
 5|xx      xxxx    xx    xx  xxxx    xx  xx
 4|xx  xx  xxxx    xx    xx  xxxx    xxxxxx
 3|xx  xx  xxxx  xxxx    xx  xxxx    xxxxxx
 2|xx  xxxxxxxxxxxxxx    xx  xxxxxx  xxxxxxxx    xxxx
 1|xxxxxxxxxxxxxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  +-------------------------------------------------
   AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
% ./histogram < /usr/share/dict/words
   |
12|
11|        xx
10|        xx
 9|xx      xx
 8|xx      xx      xx
 7|xx      xx      xx        xxxx    xx  xx
 6|xx      xx      xx    xx  xxxx    xxxxxx
 5|xx  xx  xx      xx    xx  xxxx    xxxxxx
 4|xx  xx  xx      xx    xx  xxxx    xxxxxxxx
 3|xx  xxxxxx    xxxx    xxxxxxxxxx  xxxxxxxx
 2|xxxxxxxxxx  xxxxxx    xxxxxxxxxx  xxxxxxxx      xx
 1|xxxxxxxxxxxxxxxxxx  xxxxxxxxxxx  xxxxxxxxxxx  xx
  +-------------------------------------------------
   AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
% ./histogram < /usr/include/stdio.h
   |        ^^
12|        xx
11|        xx
10|        xx                xx
 9|        xx      xx        xx
 8|        xxxx    xx        xx          xx
 7|        xxxx    xx        xx          xx
 6|      xxxxxx    xx        xx        xxxx
 5|      xxxxxx    xx        xxxx    xxxxxx
 4|      xxxxxx    xx    xx  xxxxxx  xxxxxx
 3|xx  xxxxxxxx    xx    xx  xxxxxx  xxxxxxxx    xx
 2|xx  xxxxxxxxxx  xx    xx  xxxxxx  xxxxxxxx    xx
 1|xxxxxxxxxxxxxxxxx  xxxxxxxxxxx  xxxxxxxx    xx
  +-------------------------------------------------
   AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
%
```

Δυναμική δέσμευση μνήμης

- Οι πίνακες είναι ένα πολύ χρήσιμο εργαλείο σε κάθε γλώσσα προγραμματισμού, άρα και στην C, για τη διαχείριση με ενιαίο τρόπο ενός συνόλου από ομοειδή δεδομένα. Το βασικό πρόβλημα, όμως, με τους πίνακες είναι ότι πρέπει να έχουμε ορίσει τη διάσταση τους μέσα στο πρόγραμμα, βάζοντας έτσι, κατά τη φάση της συγγραφής του προγράμματος, ένα όριο στο πλήθος των δεδομένων που μπορούμε να αποθηκεύσουμε στον πίνακα.
- Αν ορίσουμε τη διάσταση ενός πίνακα σχετικά μικρή, κάνουμε οικονομία στη μνήμη κατά την εκτέλεση του προγράμματος, αλλά αυτό έχει τότε περιορισμένες δυνατότητες. Αν ορίσουμε τη διάσταση πολύ μεγάλη, μπορεί το πρόγραμμά μας να χειριστεί και περισσότερα δεδομένα, αλλά γίνεται σπατάλη στη μνήμη όταν δεν χρειαζόμαστε να επεξεργαστούμε τόσο πολλά δεδομένα.
- Το ιδανικό θα ήταν το πρόγραμμά μας να μην δεσμεύει μνήμη εξ αρχής, αλλά να το κάνει αυτό δυναμικά, κατά τη φάση της εκτέλεσης, ανάλογα με τις ανάγκες του. Αυτό, ευτυχώς, είναι εφικτό.
- Η δυναμική δέσμευση μνήμης γίνεται σ΄ ένα χώρο που διαχειρίζεται το πρόγραμμα, ο οποίος λέγεται σωρός, και είναι διαφορετικός από το στατικό χώρο στον οποίο φυλάσσονται οι εξωτερικές μεταβλητής και τη στοίβα στην οποία φυλάσσονται οι αυτόματες μεταβλητές των συναρτήσεων.

- Στην C, η δυναμική δέσμευση μνήμης γίνεται (κυρίως) μέσω της συνάρτησης

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

- Καλώντας την `malloc` με παράμετρο έναν ακέραιο `size`,[α΄] αυτή δεσμεύει στο σωρό `size` συνεχόμενα bytes και επιστρέφει στο όνομά της ένα δείκτη στο πρώτο απ΄ αυτά.

- Ο δείκτης που επιστρέφει η `malloc` δείχνει σε τύπο `void`, δηλαδή στον κενό τύπο, αφού δεν την αφορά τι τύπου δεδομένα θα φυλάξουμε στη μνήμη που δέσμευσε.

- Αν, για κάποιο λόγο, η `malloc` δεν μπορέσει να δεσμεύσει τη μνήμη που της ζητήθηκε, επιστρέφει στο όνομά της τον κενό δείκτη `NULL`. Πάντα, όταν καλούμε την `malloc`, πρέπει να ελέγχουμε αν μας επέστρεψε δείκτη διάφορο του `NULL` και μετά να προχωρήσουμε.

- Επειδή το πλήθος των bytes που χρησιμοποιούνται για τη φύλαξη δεδομένων κάποιου τύπου μπορεί να διαφέρει μεταξύ διαφορετικών μεταγλωττιστών, λειτουργικών συστημάτων και επεξεργαστών, για να είναι τα προγράμματά μας μεταφέρσιμα, το πλήθος των bytes που ζητάμε από την `malloc` να δεσμεύσει το δίνουμε μέσω του τελεστή `sizeof`.

[α΄] Στον τυπικό ορισμό της `malloc`, ο τύπος του `size` είναι `size_t`, αλλά αυτό πρακτικά είναι `unsigned int`.

- Η έκφραση `sizeof(`⟨τύπος⟩`)` έχει σαν τιμή το πλήθος των bytes που απαιτούνται στη συγκεκριμένη υλοποίηση της C για να φυλαχθούν δεδομένα που ο τύπος τους είναι ⟨τύπος⟩. Για παράδειγμα, με το παρακάτω τμήμα προγράμματος

```
int n, *p;
.................                    /* Compute n */
p = malloc(n * sizeof(int));
if (p == NULL) {
   printf("Sorry, cannot allocate memory\n");
   return -1;
}
.................  /* Handle data starting at p */
```

  δεσμεύουμε δυναμικά χώρο στο σωρό για να φυλαχθούν `n` ακέραιοι.

- Στην προ ANSI C εποχή, ο δείκτης που επέστρεφε η `malloc` (τύπου `void *`) έπρεπε πρώτα να προσαρμοσθεί στον κατάλληλο τύπο δείκτη και μετά να ανατεθεί σε μεταβλητή. Δηλαδή την κλήση της `malloc` στο παραπάνω τμήμα προγράμματος θα την γράφαμε σαν:

```
p = (int *) malloc(n * sizeof(int));
```

  Πλέον, κάτι τέτοιο δεν είναι απαραίτητο και, για την ακρίβεια, δεν συνίσταται.

- Ο τελεστής `sizeof` μπορεί να χρησιμοποιηθεί είτε σαν `sizeof` ⟨παράσταση⟩, είτε σαν `sizeof(`⟨παράσταση⟩`)`, οπότε αυτή η έκφραση έχει σαν τιμή το πλήθος των bytes που απαιτούνται για να φυλαχθεί η ⟨παράσταση⟩, π.χ. το `sizeof buf` είναι 20, αν έχουμε ορίσει `char buf[20]`.

- ΠΡΟΣΟΧΗ! Όταν δεσμεύουμε δυναμικά μνήμη, είναι ευθύνη δική μας να την αποδεσμεύουμε όταν δεν την χρειαζόμαστε.

- Η αποδέσμευση μνήμης που έχει δεσμευθεί δυναμικά γίνεται με τη συνάρτηση

  ```
  void free(void *p)
  ```

- Η παράμετρος που περνάμε στη συνάρτηση `free` είναι ένας δείκτης που μας είχε επιστρέψει κάποια `malloc`.

- Είναι πολύ σοβαρό λάθος σ΄ ένα πρόγραμμα, να δεσμεύουμε δυναμικά μνήμη και να μην την αποδεσμεύουμε όταν δεν την χρειαζόμαστε.

- Εκτός από την `malloc`, υπάρχουν και δύο άλλες συναρτήσεις για δυναμική δέσμευση μνήμης (η `realloc` και η `calloc`). Είναι πιο σπάνια χρησιμοποιούμενες από την `malloc`. Μέσω της εντολής `man`, μπορεί να μάθει κανείς περισσότερες πληροφορίες γι΄ αυτές.

- Όταν σ΄ ένα πρόγραμμα χρησιμοποιούμε συναρτήσεις για δυναμική δέσμευση και αποδέσμευση μνήμης, πρέπει να έχουμε συμπεριλάβει στην αρχή το αρχείο επικεφαλίδας `stdlib.h`. Δηλαδή:

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

- Σε σχέση με τα προβλήματα που δημιουργούνται όταν ορίζουμε με στατικό τρόπο πίνακες για να διαχειριστούμε ένα σύνολο από δεδομένα, αντί να κάνουμε δυναμική δέσμευση μνήμης, θα πρέπει να αναφέρουμε ότι στην C υπάρχει και η δυνατότητα να ορίσουμε πίνακες με μεταβλητή διάσταση, για παράδειγμα:

  ```
  int array[n];
  ```

  Φυσικά, πριν από τη δήλωση αυτή, στην ακέραια μεταβλητή `n` πρέπει να έχουμε δώσει τιμή.

- Ο βασικός περιορισμός που υπάρχει στη δήλωση πινάκων με διάσταση που δεν είναι γνωστή κατά τη φάση συγγραφής του προγράμματος, είναι ότι μπορεί να γίνει για πίνακες που είναι αυτόματοι, δηλαδή τοπικοί μέσα σε συναρτήσεις. Δεν μπορεί να γίνει για πίνακες που θέλουμε να τους ορίσουμε στον στατικό χώρο, εξωτερικά των συναρτήσεων.

- Σε κάθε περίπτωση, είναι καλύτερη προγραμματιστική τακτική, όταν θέλουμε να δεσμεύσουμε δυναμικά μνήμη, αυτό να γίνεται μέσω `malloc` και δεικτών, παρά με πίνακες που η διάστασή τους είναι παραμετρική.

- Επίσης, να επισημανθεί ότι η δυνατότητα για πίνακες με παραμετρική διάσταση προστέθηκε στο στάνταρντ της C το 1999, οπότε παλιότεροι μεταγλωττιστές δεν το δέχονται.

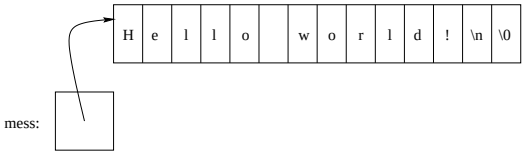
Συμβολοσειρές

- Μία συμβολοσειρά ή αλφαριθμητικό είναι ένας πίνακας από χαρακτήρες (δεδομένα τύπου `char`). Ο χαρακτήρας `'\0'`, σαν στοιχείο του πίνακα, δείχνει το τέλος της συμβολοσειράς και είναι απολύτως απαραίτητος για να μπορούν να λειτουργήσουν οι συναρτήσεις βιβλιοθήκης για το χειρισμό συμβολοσειρών.

- Λόγω της άμεσης σχέσης μεταξύ πινάκων και δεικτών, σε μία συμβολοσειρά μπορούμε να αναφερθούμε και με ένα δείκτη (τύπου `char *`) που δείχνει στον πρώτο χαρακτήρα της συμβολοσειράς.

- Μέσα σ΄ ένα πρόγραμμα, μπορούμε μ΄ ένα σύντομο τρόπο να αναφερθούμε σε μία συμβολοσειρά για την οποία γνωρίζουμε τους χαρακτήρες που περιέχει, περικλείοντας αυτούς τους χαρακτήρες μέσα σε " (χωρίς το `'\0'`). Παράδειγμα:

```
char *mess = "Hello world!\n";
```



- Οι συμβολοσειρές που περιέχονται μέσα σ΄ ένα πρόγραμμα φυλάσσονται στον ίδιο χώρο με το πρόγραμμα και, γι΄ αυτόν το λόγο, δεν είναι δυνατόν να μεταβληθούν. Για τη διαχείρισή τους, ο μεταγλωττιστής αντιστοιχεί σε κάθε τέτοια συμβολοσειρά τη διεύθυνση του πρώτου χαρακτήρα της.

- Εκτός από την αρχικοποίηση ενός δείκτη σε `char` με μία συμβολοσειρά, όπως κάναμε με την εντολή

  ```
  char *mess = "Hello world!\n";
  ```

  μπορούμε να αρχικοποιήσουμε και ένα πίνακα χαρακτήρων με μία συμβολοσειρά, ως εξής:

  ```
  char arrmess[] = "How are you?";
  ```

  Δεν υπάρχει λόγος να ορίσουμε διάσταση για τον πίνακα `arrmess`, γιατί αυτή υπολογίζεται αυτόματα από το πλήθος των χαρακτήρων της συμβολοσειράς (συν έναν, λόγω του `'\0'`).

- Μία ουσιώδης διαφορά μεταξύ των παραπάνω εντολών, είναι ότι η μεταβλητή `mess` είναι δείκτης που αρχικοποιήθηκε με τη συμβολοσειρά που θέλουμε, στη συνέχεια, όμως, μπορεί να αλλάξει τιμή, αν χρειάζεται, ενώ το όνομα του πίνακα `arrmess`, που επίσης αρχικοποιήθηκε με μία συμβολοσειρά, λειτουργεί, φυσικά, και ως δείκτης, μόνο που δεν επιτρέπεται να μεταβληθεί.

- Η δεύτερη ουσιώδης διαφορά είναι ότι δεν μπορούμε να αλλάξουμε το περιεχόμενο, δηλαδή τους χαρακτήρες, της συμβολοσειράς `"Hello world!\n"` που δείχνει αρχικά ο δείκτης `mess`, παρότι μπορούμε να αλλάξουμε την τιμή του ίδιου του δείκτη, ενώ μπορούμε να αλλάξουμε τους χαρακτήρες της συμβολοσειράς `"How are you?"`, παρότι δεν μπορούμε να αλλάξουμε το `arrmess`.

- Οι συμβολοσειρές είναι ένα πολύ χρήσιμο εργαλείο για τον προγραμματισμό στην C, γι΄ αυτό η πρότυπη βιβλιοθήκη της γλώσσας παρέχει μία σειρά από συναρτήσεις για τη διαχείρισή τους.

- Όταν χρησιμοποιούμε σ΄ ένα πρόγραμμα συναρτήσεις για διαχείριση συμβολοσειρών, πρέπει να έχουμε συμπεριλάβει στην αρχή το αρχείο επικεφαλίδας `string.h`. Δηλαδή:

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

- Μερικές από τις συναρτήσεις αυτές είναι:

```
unsigned int strlen(const char *s)
char *strcpy(char *s1, const char *s2)
int strcmp(const char *s1, const char *s2)
char *strcat(char *s1, const char *s2)
```

- Η `strlen` επιστρέφει το μήκος της συμβολοσειράς `s` (χωρίς το τελικό `'\0'`). [α΄]

- Η `strcpy` αντιγράφει τη συμβολοσειρά `s2`, μέχρι και το τελικό `'\0'`, στη συμβολοσειρά `s1`, την οποία επιστρέφει και στο όνομά της.

- Η `strcmp` συγκρίνει τις συμβολοσειρές `s1` και `s2` byte προς byte και επιστρέφει έναν ακέραιο θετικό, μηδέν ή αρνητικό, ανάλογα αν η συμβολοσειρά `s1` ακολουθεί (αλφαβητικά), ταυτίζεται ή προηγείται της συμβολοσειράς `s2`, αντίστοιχα. Η σύγκριση γίνεται με βάση τους ASCII κωδικούς των χαρακτήρων των συμβολοσειρών.

[α΄] Επισήμως ο τύπος επιστροφής της συνάρτησης είναι `size_t`, αλλά αυτός, πρακτικά, είναι `unsigned int`.

- Η `strcat` προσαρτά ένα αντίγραφο της συμβολοσειράς `s2` στο τέλος της `s1`, διαγράφοντας πρώτα το `'\0'` της `s1`, και επιστρέφει το αποτέλεσμα και στο όνομά της.
- Κάποιες από τις τυπικές παραμέτρους των συναρτήσεων για διαχείριση συμβολοσειρών έχουν δηλωθεί σαν `const` για να επισημανθεί ότι οι αντίστοιχες συμβολοσειρές δεν θα μεταβληθούν από τις συναρτήσεις.
- Οι συναρτήσεις που δημιουργούν νέες συμβολοσειρές (π.χ. `strcpy`, `strcat`) δεν αναλαμβάνουν και τη δέσμευση μνήμης για τη φύλαξή τους. Είναι ευθύνη της καλούσας συνάρτησης να το κάνει αυτό.
- Υπάρχουν και αρκετές άλλες ενδιαφέρουσες και χρήσιμες συναρτήσεις διαχείρισης συμβολοσειρών, όπως οι `strncpy`, `strchr`, `strstr`, κλπ. Η εντολή `man` είναι ένα μέσο για να μάθει κανείς περισσότερες πληροφορίες γι΄ αυτές.
- Κάποιες πιθανές υλοποιήσεις: [α΄]

```
char *strcpy(char *s1, const char *s2)
{  char *orig_s1 = s1;
   while (*s1++ = *s2++);
   return orig_s1; }

int strcmp(const char *s1, const char *s2)
{  for ( ; *s1 == *s2 ; s1++, s2++)
     if (! *s1)
       return 0;
   return *s1 - *s2; }
```

[α΄] Πώς θα υλοποιούσαμε τις `strlen` και `strcat`;

Πίνακες δεικτών και δείκτες σε δείκτες

- Όπως μπορούμε να έχουμε πίνακες ακεραίων, πίνακες πραγματικών αριθμών (κινητής υποδιαστολής, απλής ή διπλής ακρίβειας), ή πίνακες χαρακτήρων (συμβολοσειρές), έτσι μπορούμε να ορίσουμε στην C και πίνακες δεικτών.
- Με την ίδια λογική που το όνομα ενός πίνακα στοιχείων κάποιου τύπου μπορεί να θεωρηθεί (σχεδόν) ισοδύναμο με ένα δείκτη σε στοιχεία τέτοιου τύπου, έτσι και το όνομα ενός πίνακα δεικτών αντιστοιχεί σ΄ ένα δείκτη σε δείκτη σε στοιχεία του τύπου που δείχνουν τα στοιχεία του πίνακα.
- Παράδειγμα:

```
int i, j, *px[3], **ppx, s = 0;
for (i=0 ; i < 3 ; i++) {
  px[i] = malloc((i+2) * sizeof(int));
  if (px[i] == NULL)
    return -1; }
for (i=0 ; i < 3 ; i++)
  for (j=0 ; j < i+2 ; j++)
    *(px[i]+j) = i*j;
ppx = px;
for (i=0 ; i < 3 ; i++) {
  for (j=0 ; j < i+2 ; j++)
    s += *(*ppx)++;
  ppx++; }
```

Ποια θα είναι η τιμή της μεταβλητής `s` μετά την εκτέλεση των παραπάνω εντολών; [α΄]

[α΄] 15

Ορίσματα γραμμής εντολών

- Όταν εκτελούμε ένα πρόγραμμα, θέλουμε να έχουμε, κατά την κλήση του προγράμματος, τη δυνατότητα να δίνουμε στη γραμμή εντολών κάποια ορίσματα, τα οποία να μπορούν να περάσουν στο πρόγραμμα, για να τα επεξεργαστεί κατάλληλα.
- Στην C, η δυνατότητα ορισμάτων στη γραμμή εντολών παρέχεται μέσω των τυπικών παραμέτρων της συνάρτησης `main`.
- Ορίζοντας την `main` με δύο τυπικές παραμέτρους, μία ακέραια, συνήθως με όνομα `argc`, και μία πίνακα δεικτών σε χαρακτήρες (ή δείκτη σε δείκτη σε χαρακτήρες), συνήθως με όνομα `argv`, μέσα στη συνάρτηση, μπορούμε να έχουμε πρόσβαση στα ορίσματα με τα οποία κλήθηκε το πρόγραμμα στη γραμμή εντολής.
- Αν ένα εκτελέσιμο πρόγραμμα `myprog` κληθεί σαν

  ```
  % ./myprog first 241 third
  ```

  και η συνάρτηση `main` του προγράμματος έχει ορισθεί σαν

  ```
  int main(int argc, char *argv[])
  ```

  τότε, μέσα στην `main`, η μεταβλητή `argc` έχει την τιμή 4 (όσα τα ορίσματα συν το όνομα του προγράμματος) και οι δείκτες `argv[0]`, `argv[1]`, `argv[2]` και `argv[3]` δείχνουν στην αρχή των συμβολοσειρών `"./myprog"`, `"first"`, `"241"` και `"third"`, αντίστοιχα.

- Ο δείκτης `argv[argc]` (στο προηγούμενο παράδειγμα ο `argv[4]`) είναι ο κενός δείκτης, `NULL`.
- Η δεύτερη τυπική παράμετρος της `main` μπορεί να ορισθεί είτε σαν `char *argv[]` είτε σαν `char **argv`, ισοδύναμα.
- Όταν κάποιο όρισμα στη γραμμή εντολών είναι ακέραιος αριθμός, μέσα στο πρόγραμμά μας, κατά πάσα πιθανότητα, θα μας ενδιαφέρει να έχουμε διαθέσιμη την αριθμητική τιμή του αριθμού, όχι απλώς τα ψηφία του σαν μία συμβολοσειρά.
- Η μετατροπή μίας συμβολοσειράς που τα στοιχεία της είναι δεκαδικά ψηφία στην αντίστοιχη αριθμητική τιμή γίνεται με τη συνάρτηση

  ```
  int atoi(const char *s)
  ```

- Η `atoi` υπολογίζει την τιμή της (αριθμητικής) συμβολοσειράς `s` και την επιστρέφει στο όνομά της.
- Στο προηγούμενο παράδειγμα η κλήση `atoi(argv[2])` θα επέστρεφε την τιμή 241.
- Όταν χρησιμοποιούμε την `atoi`, πρέπει να κάνουμε και:

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

Πολυδιάστατοι πίνακες

- Εκτός από μονοδιάστατους πίνακες, στην C, όπως και σε όλες σχεδόν τις γλώσσες προγραμματισμού, μπορούμε να ορίσουμε και πίνακες με περισσότερες της μίας διαστάσεις.
- Με τη δήλωση

  ```
  int matrix[10][8];
  ```

  ορίζουμε ένα δισδιάστατο πίνακα με όνομα `matrix` με 10 γραμμές και 8 στήλες.
- Η αναφορά στα στοιχεία του πίνακα γίνεται με τρόπο αντίστοιχο με αυτόν των μονοδιάστατων πινάκων. Το στοιχείο `matrix[i][j]` είναι αυτό που βρίσκεται στην `i` γραμμή και στην `j` στήλη. [α΄]
- Αν έχουμε ορίσει ένα δισδιάστατο πίνακα, έστω με όνομα `matrix`, τότε το `matrix[i]` (ισοδύναμα το `*(matrix+i)`) είναι ένας δείκτης στο πρώτο στοιχείο της `i` γραμμής του πίνακα. Οπότε, το `*(matrix[i]+j)` (ισοδύναμα το `*(*(matrix+i)+j)`) δεν είναι άλλο από το στοιχείο `matrix[i][j]` του πίνακα.
- Ένας δισδιάστατος πίνακας αποθηκεύεται κατά γραμμές. Αν αναθέσουμε σ΄ ένα δείκτη `p`, που δείχνει σε τύπο ό,τι και ο τύπος των στοιχείων του πίνακα, τη διεύθυνση του πρώτου στοιχείου του πίνακα (π.χ. `matrix[0][0]`), τότε μπορούμε να προσπελάσουμε τα περιεχόμενα του πίνακα, τη μία γραμμή μετά την άλλη, με παραστάσεις της μορφής `*(p+i)`.

[α΄] Μόνο, προσοχή και εδώ, το `i` πρέπει να κυμαίνεται, στο παράδειγμά μας, από `0` έως `9` και το `j` από `0` έως `7`.

- Επίσης, αφού το `*matrix` είναι το ίδιο με το `matrix[0]`, δηλαδή ένας δείκτης στο πρώτο στοιχείο της πρώτης γραμμής (αυτής με `i=0`) του πίνακα, μπορούμε να προσπελάσουμε κατά γραμμές τα περιεχόμενα του πίνακα και με παραστάσεις της μορφής `*(*matrix+i)`.

- Φυσικά, μπορούμε να ορίσουμε και πίνακες με περισσότερες των δύο διαστάσεις, αλλά, σε πρώτη φάση, δεν έχουν ιδιαίτερη προγραμματιστική χρησιμότητα.

- Αν θέλουμε να περάσουμε σε μία συνάρτηση έναν πολυδιάστατο πίνακα, τότε στην κλήση της συνάρτησης δίνουμε το όνομά του και στον ορισμό της δηλώνουμε, μαζί με το όνομά του, και όλες τις διαστάσεις του, εκτός της πρώτης, η οποία δεν είναι υποχρεωτική. Παράδειγμα:

```
int main(void)
{  int matr[16][12];
   ....
   myfun(matr);
   ....  }

void myfun(int matr[][12])
{  ....  }
```

- Από τα προηγούμενα είναι εμφανές ότι όταν χρησιμοποιούμε πίνακες που ορίζονται στατικά, αναγκαστικά δηλώνουμε συγκεκριμένες διαστάσεις κατά τη φάση συγγραφής των προγραμμάτων μας, με αποτέλεσμα είτε να κάνουμε σπατάλη μνήμης είτε να περιοριζόμαστε στην επίλυση προβλημάτων με μικρό μέγεθος. Είναι προτιμότερο να χρησιμοποιούμε δείκτες και να κάνουμε δυναμική δέσμευση μνήμης.

- Αν σ΄ ένα πρόγραμμα χρειάζεται να φυλάξουμε δεδομένα σ΄ ένα δισδιάστατο πίνακα, αγνώστου, κατ΄ αρχήν, μεγέθους, αντί να δηλώσουμε κάτι σαν

```
int x[10000][10000];
```

μπορούμε να ορίσουμε ένα δείκτη

```
int **px;
```

και μετά να κάνουμε:

```
px = malloc(N * sizeof(int *));
if (px == NULL)
   return -1;
for (i=0 ; i < N ; i++) {
   *(px+i) = malloc(M * sizeof(int));
   if (*(px+i) == NULL)
      return -1;  }
```

- Έτσι, όταν τα N και M γίνουν γνωστά κατά τη φάση εκτέλεσης του προγράμματος, θα δεσμεύσουμε όση ακριβώς μνήμη χρειαζόμαστε. Αφού τελειώσουμε ό,τι έχουμε να κάνουμε, μετά πρέπει να αποδεσμεύσουμε τη μνήμη ως εξής:

```
for (i=0 ; i < N ; i++)
   free(*(px+i));
free(px);
```

- Για να περάσουμε σε μία συνάρτηση ένα πολυδιάστατο πίνακα ορισμένο δυναμικά, αρκεί να την καλέσουμε με παράμετρο τον δείκτη μέσω του οποίου δεσμεύτηκε η μνήμη, π.χ. `fun(px);`, και να έχουμε ορίσει τη συνάρτηση με τυπική παράμετρο ένα δείκτη του κατάλληλου τύπου. Δηλαδή:

```
void fun(int **px) { ....
```

Αρχικοποίηση πινάκων

- Όταν ορίζουμε μία μεταβλητή, μπορούμε, ως γνωστόν, να της δώσουμε και κάποια αρχική τιμή. Η δυνατότητα αυτή υπάρχει και για τους πίνακες.
- Παραδείγματα:

```
int x[7] = {5, 3, -7, 12, 0, -4, 125};
float r[] = {1.2, -4.35, 0.62e-17};
int matrix[3][5] = {
   {22, 1, -3, 8, 7},
   {-2, 0, 24, 7, -10},
   {76, 54, 12, -9, 8}
};
char arr[][2] = {{'a','b'},{'c','d'},{'e','f'}};
char mess[] = "a dog";
char buff[] = {'a',' ','d','o','g','\0'};
char *str[] = {"Hello", "world", "of", "C"};
```

- Μπορούμε, κατά τον ορισμό, να παραλείψουμε το μέγεθος ενός μονοδιάστατου πίνακα, αν αρχικοποιούμε όλα τα στοιχεία του, αφού αυτό μπορεί να προκύψει από το πλήθος των τιμών μέσα στα { και }.
- Σε πολυδιάστατους πίνακες, μπορούμε να παραλείψουμε το μέγεθος μόνο της πρώτης διάστασης.

Δείκτες σε συναρτήσεις

- Όπως μία μεταβλητή αντιστοιχεί σε μία διεύθυνση μνήμης που μπορούμε να τη φυλάξουμε σ΄ ένα δείκτη κατάλληλου τύπου, όπως επίσης μπορούμε να αναφερόμαστε σε πίνακες με τη διεύθυνση του πρώτου τους στοιχείου, έτσι μπορούμε να αναφερόμαστε και σε συναρτήσεις μέσω δεικτών.

- Ένας δείκτης σε συνάρτηση είναι μία μεταβλητή στην οποία μπορούμε να καταχωρήσουμε τη διεύθυνση της πρώτης από τις (συνεχόμενες) θέσεις μνήμης στις οποίες βρίσκεται ο κώδικας της συνάρτησης.

- Τους δείκτες συναρτήσεων μπορούμε να τους χειριστούμε περίπου όπως και τους άλλους δείκτες. Με τον τελεστή έμμεσης αναφοράς *, μπορούμε να αναφερθούμε σε συγκεκριμένες συναρτήσεις, μπορούμε να αναθέτουμε τις τιμές αυτών των δεικτών σε άλλους συμβατούς δείκτες, μπορούμε να τους περνάμε σαν παραμέτρους σε συναρτήσεις κλπ. Δεν έχει νόημα όμως να τους μεταβάλλουμε (π.χ. p++).

- Παράδειγμα:

  ```
  int (*funvar)(char *);
  ```

  Το `funvar` είναι ένας δείκτης σε συνάρτηση που επιστρέφει `int` και έχει μία τυπική παράμετρο τύπου `char *`.

- Τι διαφορά θα είχε αν γράφαμε το παρακάτω; [α΄]

  ```
  int *funvar(char *);
  ```

[α΄] Εδώ το `funvar` είναι το όνομα συγκεκριμένης συνάρτησης, όχι δείκτης σε συνάρτηση, που επιστρέφει δείκτη σε ακέραιο (`int *`) και έχει μία τυπική παράμετρο τύπου `char *`.

Διαχείριση ορισμάτων στη γραμμή εντολής

```
/* File: cmdlineargs.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int checkint(char *);
void reverse(char *);

int main(int argc, char *argv[])
{ int sum = 0;   /* Initialize sum of integers in command line */
  while (--argc) {         /* Loop while arguments are present */
    if(checkint(*++argv))           /* Is argument an integer? */
      sum += atoi(*argv); /* Then, sum it into the accumulator */
    else {
      reverse(argv[0]); /* Else, reverse argument and print it */
      printf("Reversed argument: %s\n", argv[0]);
    }
  }                 /* Finally, print sum of integer arguments */
  printf("\nSum of integers given is: %d\n", sum);
  return 0;
}

int checkint(char *s)
{ char *start;
  while(*s == ' ' || *s == '\t') s++;    /* Eat all whitespace */
  if (*s == '-' || *s == '+') s++;             /* Eat one sign */
  start = s;                    /* Mark the begining of numbers */
  while(*s >= '0' && *s <= '9') s++;        /* Eat all numbers */
  return (*s == '\0' && start != s);
             /* Return if there where numbers and nothing else */
}

void reverse(char *s)
{ char c;
  int i, j;
  for (i=0, j=strlen(s)-1 ; i < j ; i++, j--) {
    c = s[i];  /* Visit string from start and end concurrently */
    s[i] = s[j];       /* and exchange characters in symmetric */
    s[j] = c;             /* positions until middle is reached */
  }
}
```

```
% gcc -o cmdlineargs cmdlineargs.c
% ./cmdlineargs This is a test
Reversed argument: sihT
Reversed argument: si
Reversed argument: a
Reversed argument: tset

Sum of integers given is: 0
% ./cmdlineargs And 123 another 12
Reversed argument: dnA
Reversed argument: rehtona

Sum of integers given is: 135
% ./cmdlineargs minus five -5 plus twenty two 22
Reversed argument: sunim
Reversed argument: evif
Reversed argument: sulp
Reversed argument: ytnewt
Reversed argument: owt

Sum of integers given is: 17
%
```

Πρόσθεση πολυψήφιων αριθμών

```
/* File: addnumbs.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *addnumbs(char *, char *);

int main(int argc, char *argv[])
{ char *sum;
  if (argc != 3) {               /* Run with exactly two arguments */
    printf("Usage: %s <numb1> <numb2>\n", argv[0]); return 1; }
  if ((sum = addnumbs(argv[1], argv[2])) == NULL) { /* Compute sum */
    printf("Sorry, a memory problem occurred\n"); return 1; }
  printf("%s + %s = %s\n", argv[1], argv[2], sum);
  free(sum);                   /* Free memory malloc'ed by addnumbs */
  return 0;
}
```

```
char *addnumbs(char *s1, char *s2)
{ int i, j, k, d1, d2, sum, carry;
  char *s3, *tmp, *result;
  i = strlen(s1)-1;     /* Index to last character of first number */
  j = strlen(s2)-1;    /* Index to last character of second number */
  k = (i > j) ? (i+1) : (j+1);   /* Index to last character of sum */
  if ((s3 = malloc((k+2)*sizeof(char))) == NULL)
    return NULL;                      /* Allocate memory for result */
  s3[k+1] = '\0';        /* Proper termination of resulting string */
  carry = 0;                          /* Initial carry for addition */
  for ( ; k >= 0 ; i--, j--, k--) {
                    /* Loop till first digit of result is computed */
    d1 = (i >= 0) ? (s1[i]-'0') : 0;       /* Get i-th digit of s1 */
    d2 = (j >= 0) ? (s2[j]-'0') : 0;       /* Get j-th digit of s2 */
    sum = d1+d2+carry;                           /* Sum two digits */
    carry = sum/10;                                 /* Next carry */
    s3[k] = sum%10+'0';                /* Put k-th digit of result */
  }
  tmp = s3;                    /* Save s3 for freeing it afterwards */
  while (*s3 == '0')                /* Eat leading 0s in the result */
    s3++;
  if(*s3 == '\0')                   /* At least one digit is needed */
    s3--;
  if ((result = malloc((strlen(s3)+1)*sizeof(char))) == NULL)
    return NULL;  /* Allocate memory for result without leading 0s */
  strcpy(result, s3);                /* Copy corrected s3 to result */
  free(tmp);                                    /* Free original s3 */
  return result;
}
```

```
% gcc -o addnumbs addnumbs.c
% ./addnumbs 12345 67890
12345 + 67890 = 80235
% ./addnumbs 125 88275346771923625166
125 + 88275346771923625166 = 88275346771923625291
% ./addnumbs 999999999999999 999999
999999999999999 + 999999 = 1000000000999998
% ./addnumbs 0000000000023 00000057
0000000000023 + 00000057 = 80
% ./addnumbs 3252352362364362362366 3262363262362363622
3252352362364362362366 + 3262363262362363622 = 3255614725626724725988
% ./addnumbs 0000000 0000000000
0000000 + 0000000000 = 0
%
```

Απαριθμήσεις

- Στην C παρέχεται ένας βολικός τρόπος σύνδεσης σταθερών ακέραιων τιμών με ονόματα, μέσω των απαριθμήσεων. Πρόκειται για μία δυνατότητα αντίστοιχη του `#define`, μόνο που την διαχειρίζεται ο μεταγλωττιστής, όχι ο προεπεξεργαστής.

- Παράδειγμα:

```
enum boolean {NO, YES};
```

  Για τη συνέχεια, το `NO` ταυτίζεται με το 0 και το `YES` με το 1.

- Μπορούμε να δίνουμε συγκεκριμένες τιμές στις σταθερές απαρίθμησης, υπονοώντας ότι οι επόμενες σταθερές έχουν συνεχόμενες τιμές, εκτός αν και σε αυτές αποδίδονται τιμές.

- Παραδείγματα:

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC};
enum escapes {BELL = '\a', BACKSPACE = '\b',
              TAB = '\t', NEWLINE = '\n'};
```

  Εδώ, οι σταθερές για τους μήνες στην απαρίθμηση `months` αντιστοιχούν στους αριθμούς από 1 έως 12, ενώ οι σταθερές της απαρίθμησης `escapes` αντιστοιχούν στις συγκεκριμένες τιμές που έχουν καθορισθεί για την καθεμία.

- Αναφορά στις απαριθμήσεις, γίνεται στην παράγραφο §2.3 του [KR].

Δομές

- Πολλές φορές, μία σύνθετη οντότητα μπορεί να καθορισθεί από ένα σύνολο δεδομένων, πιθανώς διαφορετικών τύπων, οπότε θα ήταν χρήσιμο να ομαδοποιούσαμε τα δεδομένα αυτά κάτω από ένα γενικό όνομα, με σκοπό να αναφερόμαστε στην οντότητα αυτή με το όνομα αυτό.
- Στην C, η δυνατότητα ομαδοποίησης δεδομένων, που αποτελούν κατά κάποιο τρόπο τα χαρακτηριστικά μίας προγραμματιστικής οντότητας, παρέχεται από τις δομές.
- Μία δομή ορίζεται με τον εξής τρόπο:
  `struct` ⟨ετικέτα δομής⟩ {
  ⟨τύπος⟩$_1$ ⟨μέλος⟩$_1$;
  ⟨τύπος⟩$_2$ ⟨μέλος⟩$_2$;
  .....
  ⟨τύπος⟩$_n$ ⟨μέλος⟩$_n$;
  };
  Ο ορισμός μίας δομής αποτελείται από τους ορισμούς των μελών της, που περιγράφονται σαν μεταβλητές συγκεκριμένων τύπων.
- Φυσικά, πίνακες, δείκτες ή ακόμα και δομές μπορούν να είναι μέλη μίας δομής.
- Για πιο αναλυτική συζήτηση περί δομών, παραπέμπεστε στις παραγράφους §6.1 έως §6.6 του [KR]. Για δημιουργία νέων ονομάτων τύπων (εντολή `typedef`), ενώσεις και πεδία bit, που θα συζητηθούν στη συνέχεια, οι παράγραφοι από το [KR] είναι οι §6.7, §6.8 και §6.9, αντίστοιχα.

- Παραδείγματα:

```
struct employee {
  char firstname[10];
  char lastname[18];
  int id_number;
  float salary;
};
struct wordinfo {
  char *word;
  int nlines;
};
```

- Στα παραδείγματα αυτά ορίζουμε μία δομή `struct employee` για την αναπαράσταση πληροφοριών για ένα υπάλληλο (μικρό όνομα, επώνυμο, αριθμός μητρώου και μισθός) και μία δομή `struct wordinfo` για την αναπαράσταση της πληροφορίας σχετικά με την ύπαρξη μίας λέξης σε κάποιο κείμενο, καθώς και το πλήθος των γραμμών στις οποίες εμφανίζεται.

- Ο ορισμός μίας δομής είναι μία δήλωση για την οποία δεν γίνεται κάποια δέσμευση μνήμης. Μπορούμε όμως να ορίσουμε συγκεκριμένες μεταβλητές που έχουν σαν τύπο μία δομή, ως εξής:
`struct` ⟨ετικέτα δομής⟩ ⟨μεταβλητή⟩$_1$, ..., ⟨μεταβλητή⟩$_k$;
Τότε, δεσμεύεται η απαιτούμενη μνήμη για τις μεταβλητές, ανάλογα με τον χώρο που χρειάζεται για να φυλαχθούν τα μέλη της δομής.

- Παραδείγματα:

```
struct employee jim, jane, the_employee;
struct wordinfo first_word, next_word, last_word;
```

- Άλλα παραδείγματα δηλώσεων δομών:

```
struct point {
  double x;
  double y;
};
struct upright_rectangle {
  struct point p1;
  struct point p2;
};
```

- Εδώ ορίζουμε τη δομή `struct point` για την αναπαράσταση ενός σημείου στο επίπεδο (μέσω των συντεταγμένων του), καθώς και τη δομή `struct upright_rectangle` για την αναπαράσταση ενός ορθογωνίου παραλληλογράμμου στο επίπεδο, με τις πλευρές παράλληλες στους άξονες (μέσω δύο σημείων που είναι απέναντι κορυφές σε μία διαγώνιο). [α΄]

[α΄] Μπορείτε να ορίσετε δομές και για άλλες γεωμετρικές οντότητες στο επίπεδο, για παράδειγμα, ευθύγραμμα τμήματα, τρίγωνα, κύκλους, ή, ανεξαρτήτως προσανατολισμού, τετράγωνα, ρόμβους, ορθογώνια παραλληλόγραμμα, (πλάγια) παραλληλόγραμμα και τυχαία τετράπλευρα; Μπορείτε να ορίσετε και γεωμετρικές οντότητες στον τρισδιάστατο χώρο;

- Ο ορισμός μεταβλητών με τύπο κάποια συγκεκριμένη δομή μπορεί να γίνει ταυτόχρονα με τη δήλωση της δομής. Παράδειγμα:

```
struct point {
  double x;
  double y;
} pa, pb, pc;
```

  Στην περίπτωση αυτή, δεν είναι απαραίτητο να δώσουμε ετικέτα στη δομή. Για παράδειγμα, ο εξής ορισμός των μεταβλητών `pa`, `pb` και `pc` είναι αποδεκτός:

```
struct {
  double x;
  double y;
} pa, pb, pc;
```

  Φυσικά, τότε, αν θέλουμε στη συνέχεια να ορίσουμε και άλλες μεταβλητές του τύπου της δομής, πρέπει να επαναλάβουμε τη δήλωσή της, αφού δεν της είχαμε δώσει κάποια ετικέτα.

- Μεταβλητές με τύπο δομή μπορούν να ανατίθενται σαν τιμές σε άλλες μεταβλητές του ίδιου τύπου, μπορούν να δίνονται σαν ορίσματα κατά την κλήση συναρτήσεων και μπορούν να επιστρέφονται από συναρτήσεις στο όνομά τους.

- Με τον ορισμό μίας μεταβλητής τύπου δομής, μπορούμε να κάνουμε και αρχικοποίηση των μελών της. Παράδειγμα:

```
struct point my_point = {22.4, -38.9};
```

  Για αυτόματες μεταβλητές τύπου δομής, αλλά και προκαθορισμένων τύπων, αρχικοποίηση μπορεί να γίνει και με εντολή αντικατάστασης ή μέσω επιστροφής συνάρτησης.

- Η αναφορά σ΄ ένα μέλος μίας μεταβλητής τύπου δομής γίνεται με την παράσταση
  ⟨μεταβλητή⟩ . ⟨μέλος⟩

- Παραδείγματα:

```
struct point vert1, vert2;
struct upright_rectangle my_rect;
vert1.x = 2.4;
vert2.y = 7.8;
my_rect.p1.x = -8.3;
```

  Η τελευταία εντολή είναι ισοδύναμη με την

```
(my_rect.p1).x = -8.3;
```

  επειδή ο τελεστής . είναι αριστερά προσεταιριστικός.

- Όπως ορίζουμε μεταβλητές με τύπο κάποια δομή, έτσι μπορούμε να ορίσουμε και δείκτες σε δομές. Παράδειγμα:

  ```
  struct point *ppa, *ppb;
  ```

  Φυσικά, είναι δική μας ευθύνη να κάνουμε τους δείκτες `ppa` και `ppb` να δείξουν σε διευθύνσεις στις οποίες φυλάσσονται ή θα φυλαχθούν δεδομένα τύπου δομής. Αυτό μπορεί να γίνει είτε με την ανάθεση της διεύθυνσης κάποιας ήδη ορισμένης μεταβλητής τύπου δομής, είτε με δυναμική δέσμευση. Παραδείγματα:

  ```
  struct point my_point;
  ppa = &my_point;
  ppb = malloc(sizeof(struct point));
  ```

- Έχοντας ορίσει ένα δείκτη σε δομή, όπως τον `ppa` στο προηγούμενο παράδειγμα, μπορούμε να αναφερθούμε σε συγκεκριμένο μέλος κατά τα γνωστά, για παράδειγμα `(*ppa).x`.

- Στην παράσταση `(*ppa).x` οι παρενθέσεις είναι απαραίτητες. Αν γράφαμε `*ppa.x`, αυτό, λόγω των σχετικών προτεραιοτήτων των τελεστών `.` και `*` θα ήταν ισοδύναμο με το `*(ppa.x)`, το οποίο όμως, στην προκείμενη περίπτωση δεν είναι συντακτικά σωστό, αφού το `ppa.x` δεν είναι δείκτης.

- Αν ένας ⟨δείκτης⟩ δείχνει σε κάποια δομή και θέλουμε να αναφερθούμε σ΄ ένα ⟨μέλος⟩ της, αντί να γράφουμε
  (*⟨δείκτης⟩) .⟨μέλος⟩
  η C μας δίνει τη δυνατότητα να γράψουμε πιο απλά
  ⟨δείκτης⟩->⟨μέλος⟩

- Όπως ο τελεστής ., έτσι και ο -> είναι αριστερά προσεταιριστικός. Για παράδειγμα, αν έχουμε ορίσει

  ```
  struct upright_rectangle *rp;
  ```

  τότε η παράσταση `rp->p1.x` είναι η ίδια με την `(rp->p1).x`.

- Οι τελεστές . και -> μαζί με τις () για κλήσεις συναρτήσεων και τις [] για δείκτες πινάκων βρίσκονται στην κορυφή της ιεραρχίας προτεραιότητας των τελεστών και επομένως συνδέονται ισχυρά με τους τελεστέους.

- Παράδειγμα:

  ```
  struct wordinfo *pw;
  ```

  Με την έκφραση `++pw->nlines` αυξάνει το `nlines` γιατί είναι ισοδύναμη με την `++(pw->nlines)`. Αν θέλαμε να αυξήσουμε τον δείκτη `pw` θα έπρεπε να γράψουμε `(++pw)->nlines` ή `(pw++)->nlines`[α΄], ανάλογα με το πότε θα επιθυμούσαμε να γίνει η αύξηση του δείκτη, πριν ή μετά την προσπέλαση του μέλους `nlines`.

- Έχοντας πλέον αναφερθεί σε όλους τους τελεστές της C, ο Πίνακας 2–1 του [KR] (σελ. 83), είναι μία πολύ χρήσιμη πηγή για αναφορά, σχετικά με τις προτεραιότητές τους.

[α΄] Εδώ δεν είναι απαραίτητες οι παρενθέσεις, αφού δεν θα υπήρχε αμφιβολία για το τι θα σήμαινε το `pw++->nlines`.

- Όπως συμβαίνει και με όλους τους τύπους στην C, μπορούμε να ορίσουμε και πίνακες δομών.
- Παράδειγμα:

```
int main(void)
{ int i, n;
  struct worddata {
    char *word;
    int numb;
    char let;
  } wordarray[] = {
      {"ABCD", 1, 'a'}, {"EF", 2, 'b'},
      {"GHIJ", 3, 'c'}, {"KL", 4, 'd'},
      {"M", 5, 'e'}};
  struct worddata *p = wordarray;
  n = sizeof(wordarray)/sizeof(struct worddata);
  printf("%d ", n);
  for (i=0 ; i < 2 ; i++) {
    printf("%c ", *p->word++);
    printf("%s ", ++p->word);
    printf("%c ", p->let++);
    printf("%c ", (++p)->let);
    printf("%d ", p++->numb);
    printf("%d ", ++p->numb); }
  for (i=0 ; i < n ; i++)
    printf("%s %d %c ", wordarray[i].word,
           wordarray[i].numb, wordarray[i].let);
  printf("\n"); return 0; }
```

Τι θα εκτυπωθεί από το πρόγραμμα αυτό; [α΄]

[α΄] `5 A CD a b 2 4 G IJ c d 4 6 CD 1 b EF 2 b IJ 4 d KL 4 d M 6 e`

Εύρεση τριγώνων μεγίστου και ελαχίστου εμβαδού

```
/* File: triangles.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

struct point {
  double x;                  /* A point in 2 dimensions is defined */
  double y;                        /* by its x- and y- coordinates */
} *points;

struct triangle {
  struct point a;   /* A triangle is defined by its three vertices */
  struct point b;
  struct point c;
};

struct triangle get_triangle(int, int, int);
double triangle_area(struct triangle *);
double get_side(struct point, struct point);

int main(int argc, char *argv[])
{ int i, j, k, n = 10;
  long seed;
  struct triangle tr, max_tr, min_tr;
  double area, max_area, min_area;
  seed = time(NULL);
  if (argc > 1)
    n = atoi(argv[1]);
  if (n < 3) {           /* In order to have at least one triangle */
    printf("At least three points are needed\n");
    return 1;
  }
  if ((points = malloc(n * sizeof(struct point))) == NULL) {
                              /* Allocate memory to store n points */
    printf("Sorry, not enough memory!\n");
    return 1;
  }
```

```
                              /* Initialize random number generator */
  srand((unsigned int) seed);
  for (i=0 ; i < n ; i++) {
         /* Generate points with coordinates between 0.0 and 100.0 */
/* RAND_MAX is max number that rand() returns - usually 2147483647 */
    (points+i)->x = (100.0 * rand())/(RAND_MAX+1.0);
    (points+i)->y = (100.0 * rand())/(RAND_MAX+1.0);
  }
  for (i=0 ; i < n ; i++)              /* Printout points generated */
    printf("P%-2d: (%4.1f,%4.1f)\n", i, (points+i)->x, (points+i)->y);
  tr = get_triangle(0, 1, 2);                  /* Get first triangle */
  area = triangle_area(&tr);     /* Compute area of first triangle */
           /* Initialize max and min triangles with first triangle */
  max_tr = min_tr = tr;
  max_area = min_area = area;    /* Areas of max and min triangles */
  for (i=0 ; i < n-2 ; i++)    /* Iterate through all combinations */
    for (j=i+1 ; j < n-1 ; j++)  /* of points in order to form all */
      for (k=j+1 ; k < n ; k++) {             /* possible triangles */
        tr = get_triangle(i, j, k);         /* Get current triangle */
        area = triangle_area(&tr);      /* Area of current triangle */
        if (area > max_area) {       /* Is current larger than max? */
          max_tr = tr;
          max_area = area;
        }
        if (area < min_area) {      /* Is current smaller than min? */
          min_tr = tr;
          min_area = area;
        }
      }
  printf("\n");
  printf("Max triangle: ");                /* Printout max triangle */
  printf("(%4.1f,%4.1f) ", max_tr.a.x, max_tr.a.y);
  printf("(%4.1f,%4.1f) ", max_tr.b.x, max_tr.b.y);
  printf("(%4.1f,%4.1f) ", max_tr.c.x, max_tr.c.y);
  printf(" Area: %10.5f\n", max_area);
  printf("Min triangle: ");                /* Printout min triangle */
  printf("(%4.1f,%4.1f) ", min_tr.a.x, min_tr.a.y);
  printf("(%4.1f,%4.1f) ", min_tr.b.x, min_tr.b.y);
  printf("(%4.1f,%4.1f) ", min_tr.c.x, min_tr.c.y);
  printf(" Area: %10.5f\n", min_area);
  return 0;
}
```

```
struct triangle get_triangle(int i, int j, int k)
{ struct triangle tr;
  tr.a = *(points+i);
  tr.b = *(points+j);
  tr.c = *(points+k);
  return tr;              /* Return triangle with vertices i, j, k */
}

double triangle_area(struct triangle *tr)
{ double s1, s2, s3, t;
  s1 = get_side(tr->a, tr->b);            /* Get length of side ab */
  s2 = get_side(tr->b, tr->c);            /* Get length of side bc */
  s3 = get_side(tr->c, tr->a);            /* Get length of side ca */
  t = (s1+s2+s3)/2;               /* Compute half of the perimeter */
  return sqrt(t*(t-s1)*(t-s2)*(t-s3));             /* Return area */
}

double get_side(struct point p1, struct point p2)
{            /* Return Euclidean distance between points p1 and p2 */
  return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
```

```
% gcc -o triangles triangles.c -lm
% ./triangles 17
P0 : (37.5,51.6)
P1 : (31.4, 0.8)
P2 : (10.3,32.8)
P3 : ( 4.5,38.4)
P4 : (27.4,96.3)
P5 : (28.4,43.0)
P6 : (69.8,55.7)
P7 : (13.3,71.4)
P8 : (17.8,76.2)
P9 : ( 3.9, 8.8)
P10: (29.6,69.6)
P11: (20.6,43.3)
P12: ( 9.1,76.1)
P13: (33.9,25.6)
P14: (34.1,66.4)
P15: (15.1,22.9)
P16: (67.5,49.7)

Max triangle: (27.4,96.3) (69.8,55.7) ( 3.9, 8.8)  Area: 2331.69411
Min triangle: (28.4,43.0) (17.8,76.2) (33.9,25.6)  Area:    0.09948
%
```

Αυτο-αναφορικές δομές

- Τα μέλη μίας δομής μπορεί να είναι οποιουδήποτε τύπου, ακόμα και δείκτες σε δομές του ίδιου τύπου. Χρησιμοποιώντας τέτοιου είδους δομές, που ονομάζονται αυτο-αναφορικές και είναι πολύ χρήσιμες στον προγραμματισμό, μπορούμε να οργανώσουμε δεδομένα με τρόπους που διευκολύνουν εξαιρετικά τη διαχείριση και επεξεργασία τους.

- Οι πλέον συνήθεις οργανώσεις δεδομένων μέσω αυτο-αναφορικών δομών είναι οι συνδεδεμένες λίστες (ή, απλά, λίστες) και τα δυαδικά δέντρα. Μπορεί κανείς να ορίσει και άλλες οργανώσεις δεδομένων, όπως διπλά συνδεδεμένες λίστες, ουρές, δέντρα όχι κατ΄ ανάγκη δυαδικά, κλπ.

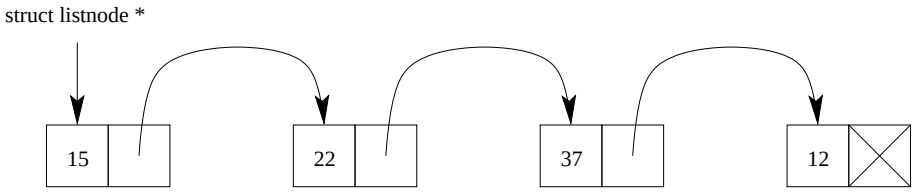
- Έστω η δήλωση:

```
struct listnode {
  int value;
  struct listnode *next;
};
```

  Αυτή η δομή ορίζει έναν κόμβο λίστας στον οποίο φυλάσσεται ένας ακέραιος και ένας δείκτης σε κόμβο λίστας.

- Αν θέλουμε να αποθηκεύσουμε μία ακολουθία από ακεραίους, απροσδιορίστου πλήθους, αλλά και μεταβαλλόμενου κατά τη διάρκεια της εκτέλεσης ενός προγράμματος, μπορούμε να το κάνουμε βάζοντας κάθε ακέραιο σ΄ ένα κόμβο λίστας και δείχνοντας από κάθε κόμβο στον επόμενό του.

- Η αναφορά στη λίστα γίνεται με ένα δείκτη στον πρώτο κόμβο της. Ο τελευταίος κόμβος της λίστας έχει σαν δείκτη σε επόμενο κόμβο το `NULL`.

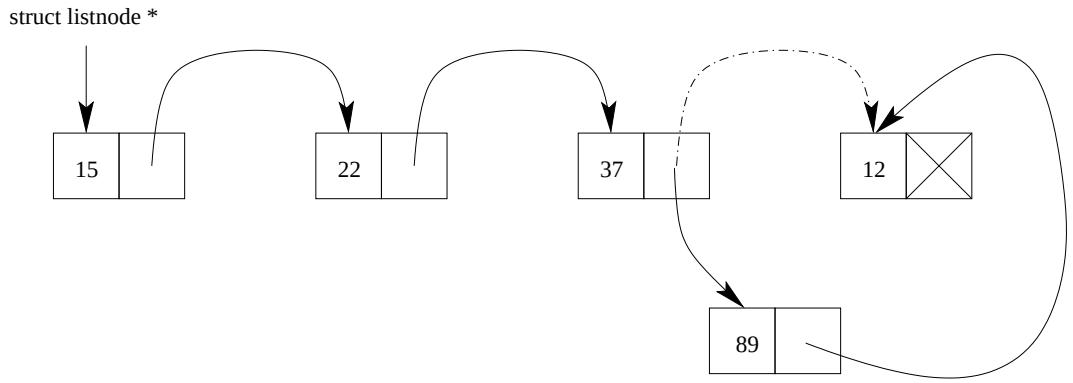


  Αυτή είναι μία λίστα με στοιχεία, κατά σειρά, τα 15, 22, 37 και 12.

- Για να υλοποιήσουμε τη δυναμική φύση της λίστας, πρέπει πάντα να γίνεται, για τη φύλαξη ενός κόμβου της, η κατάλληλη δυναμική δέσμευση μνήμης (μέσω `malloc`) και, φυσικά, η αποδέσμευση (μέσω `free`), όταν δεν χρειαζόμαστε πλέον τον κόμβο.

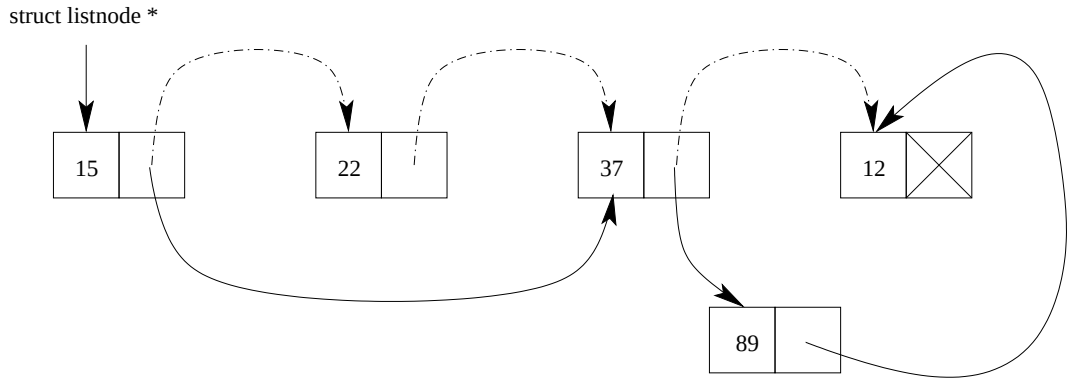
- Οι κόμβοι μίας λίστας δεν φυλάσσονται σε διαδοχικές θέσεις μνήμης, αφού η μνήμη γι΄ αυτούς δεσμεύεται με διαφορετικές `malloc`. Αυτό σημαίνει ότι δεν μπορούμε να έχουμε άμεση πρόσβαση στο N-οστό στοιχείο μίας λίστας, όπως στους πίνακες, αλλά μόνο ακολουθώντας την αλυσίδα των στοιχείων από το πρώτο έως το N-οστό.

- Η παρεμβολή ενός κόμβου σε συγκεκριμένη θέση σε μία λίστα δεν απαιτεί την μετακίνηση κανενός άλλου κόμβου. Απλώς πρέπει να τροποποιηθεί ο δείκτης για επόμενο του κόμβου που προηγείται της θέσης που εισάγεται ο νέος κόμβος, ώστε να δείξει πλέον στον νέο κόμβο. Επίσης, ο δείκτης για επόμενο του νέου κόμβου θα πρέπει να δείξει στον επόμενο κόμβο από τη θέση που έγινε η εισαγωγή.



Εδώ, έγινε εισαγωγή, στη λίστα της προηγούμενης σελίδας, του στοιχείου 89 μεταξύ των στοιχείων 37 και 12.

- Αντίστοιχα ισχύουν και για τη διαγραφή ενός κόμβου από μία λίστα. Δεν απαιτείται καμία μετακίνηση.



Εδώ, έγινε διαγραφή του στοιχείου 22 από την προηγούμενη λίστα.

- Φυσικά, μπορούμε να ορίσουμε κόμβους λίστας στους οποίους η πληροφορία που φυλάσσουμε (πλην του δείκτη στον επόμενο κόμβο) να είναι οποιουδήποτε τύπου, όχι μόνο ακέραιος, αλλά και κινητής υποδιαστολής, δείκτης σε κάποιο τύπο (π.χ. συμβολοσειρά), γενικά οτιδήποτε.

- Επίσης, μπορούμε σ΄ ένα κόμβο λίστας να ορίσουμε περισσότερα από ένα μέλη για την πληροφορία του κόμβου, για παράδειγμα, μία συμβολοσειρά, έναν ακέραιο και ένα δείκτη σε λίστα. [α΄]

- Μία άλλη, πολύ χρήσιμη, οργάνωση δεδομένων μέσω αυτο-αναφορικών δομών είναι τα δυαδικά δέντρα.

- Ένα δυαδικό δέντρο είναι μία συλλογή κόμβων, οργανωμένων σε μία δενδρική διάταξη, καθένας από τους οποίους μπορεί να έχει μέχρι δύο κόμβους-παιδιά. Κάθε κόμβος έχει έναν κόμβο-γονέα, εκτός από τον πρώτο κόμβο του δέντρου, που είναι ο κόμβος-ρίζα.

- Τα παιδιά ενός κόμβου αναφέρονται σαν το αριστερό και το δεξί παιδί του κόμβου. Για την αναπαράσταση ενός κόμβου χρησιμοποιούμε μία αυτο-αναφορική δομή, όπως η εξής:

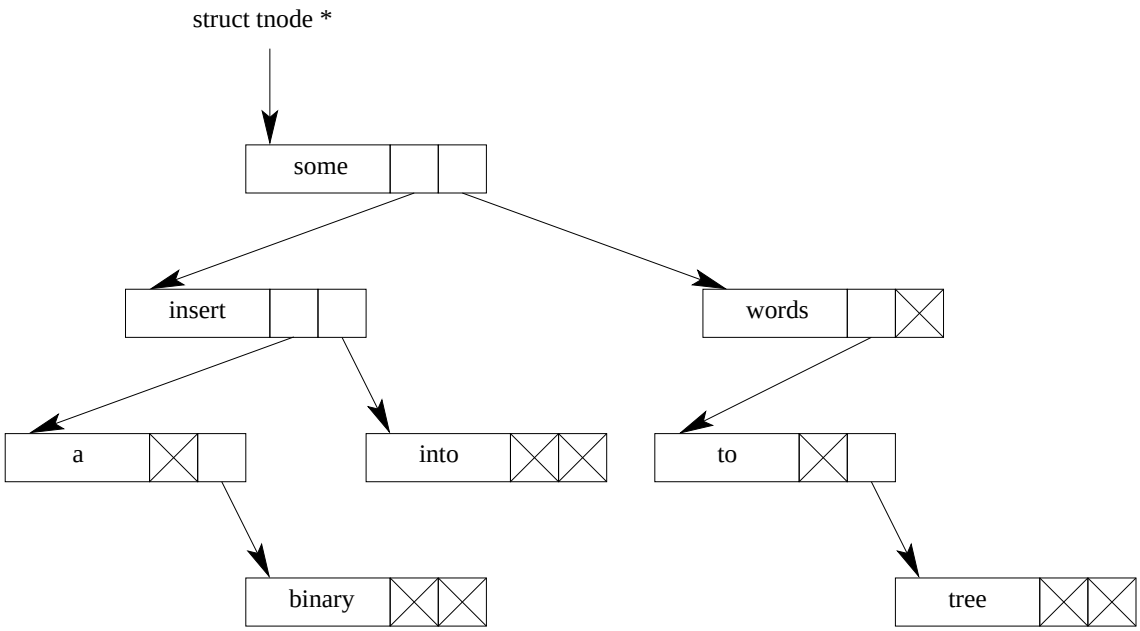
```
struct tnode {
  int value;
  struct tnode *left;
  struct tnode *right;
};
```

[α΄] Δεν είναι ενδιαφέρον να μπορούμε να έχουμε και λίστες από λίστες;

- Με τη χρήση της δομής `struct tnode`, σε κάθε κόμβο του δέντρου φυλάσσεται ένας ακέραιος, αλλά θα μπορούσαμε σαν πληροφορία προς φύλαξη να έχουμε δεδομένα οποιουδήποτε τύπου, όπως και στις λίστες, και, φυσικά, και περισσότερα του ενός δεδομένα. Οι δείκτες `left` και `right` είναι οι διευθύνσεις των κόμβων που είναι αριστερό και δεξί παιδί, αντίστοιχα, του κόμβου.
- Όπως και στις λίστες, αν κάποιο παιδί δεν υπάρχει, ο αντίστοιχος δείκτης είναι `NULL`.
- Η αναφορά σ΄ ένα δυαδικό δέντρο γίνεται μέσω ενός δείκτη στη ρίζα του.
- Κατά τη δημιουργία ενός δυαδικού δέντρου, πρέπει να δεσμεύεται δυναμικά μνήμη (μέσω `malloc`) για να φυλαχθεί κάθε κόμβος, ενώ όταν ένας κόμβος δεν χρειάζεται πλέον, πρέπει η μνήμη που καταλαμβάνει να αποδεσμεύεται (μέσω `free`).
- Τα δυαδικά δέντρα είναι πραγματικά χρήσιμα ως οργανώσεις δεδομένων όταν είναι ταξινομημένα. Ένα δυαδικό δέντρο είναι ταξινομημένο όταν, με βάση κάποιο κριτήριο διάταξης στην πληροφορία που αποθηκεύουμε στους κόμβους του, ο κόμβος-ρίζα έπεται όλων των κόμβων στο αριστερό παιδί του και προηγείται όλων των κόμβων στο δεξί παιδί του και, επίσης, τόσο το αριστερό όσο και το δεξί παιδί είναι ταξινομημένα με την ίδια λογική.

- Σαν κριτήρια διάταξης της πληροφορίας των κόμβων ενός δέντρου, μπορούμε να έχουμε την αριθμητική διάταξη, αν η πληροφορία αυτή είναι ένας ακέραιος ή ένας πραγματικός αριθμός, ή την αλφαβητική διάταξη, αν η πληροφορία είναι μία συμβολοσειρά. Αν έχουμε περισσότερες της μίας πληροφορίες στον κόμβο, τότε κάποια, δηλαδή το αντίστοιχο μέλος της δομής, παίζει τον ρόλο του λεγόμενου κλειδιού, που με βάση αυτό ορίζεται η διάταξη μεταξύ δύο κόμβων.

- Έστω η πρόταση "some words to insert into a binary tree". Στο σχήμα, φαίνεται ένα ταξινομημένο δυαδικό δέντρο που περιέχει τις λέξεις της πρότασης αυτής στους κόμβους του.



- Το συγκεκριμένο δέντρο κατασκευάστηκε εισάγοντας σταδιακά τις λέξεις της πρότασης με τη σειρά που εμφανίζονται στην πρόταση. Αν είχαν εισαχθεί με διαφορετική σειρά, το δέντρο θα ήταν τελικά διαφορετικό. [α΄]

[α΄] Ποιο θα ήταν το δέντρο αν η εισαγωγή των λέξεων γινόταν από την τελευταία λέξη της πρότασης προς την πρώτη;

Δημιουργία νέων ονομάτων τύπων

- Στην C παρέχεται η δυνατότητα, μέσω της εντολής `typedef`, να δώσουμε δικά μας ονόματα σε τύπους που χρησιμοποιούμε συχνά, και μετά να ορίζουμε μεταβλητές αυτών των τύπων με βάση το νέο όνομα.
- Παραδείγματα:

```
typedef int Length;
typedef char *String;

typedef struct listnode *Listptr;

struct listnode {
  int value;
  Listptr next;
};

typedef struct tnode *Treeptr;

typedef struct tnode {
  int value;
  Treeptr left;
  Treeptr right;
} Treenode;

Length len, maxlen;
String name, line[10];
Listptr a_list;
Treenode a_tree_node;
Treeptr a_tree;
```

Ενώσεις και πεδία bit

- Όταν η μνήμη ήταν ακριβή, είχε νόημα στα προγράμματά μας να κάνουμε οικονομία μνήμης.
- Ένας τρόπος για εξοικονόμηση μνήμης είναι οι ενώσεις, που ορίζονται όπως οι δομές (με τη λέξη-κλειδί `union`). Δεν δεσμεύεται χώρος για όλα τα μέλη της όταν ορίζεται μία μεταβλητή τύπου ένωσης, αλλά αυτός που απαιτείται για τη φύλαξη του μέλους με τις μεγαλύτερες απαιτήσεις σε μνήμη. Όλα τα μέλη φυλάσσονται σ΄ αυτόν τον χώρο. Παράδειγμα:

```
union alternative_data {
  int selection;
  int ivalue;
  float fvalue;
  char *svalue;
} id_numb;
```

- Οι ενώσεις είναι χρήσιμες και για το πακετάρισμα τυπικών παραμέτρων συναρτήσεων, όταν αυτές δεν χρειάζονται όλες ταυτόχρονα κατά την κλήση της συνάρτησης.
- Άλλος τρόπος για οικονομία μνήμης είναι τα πεδία bit. Ορίζοντας μία δομή, μπορούμε να δηλώσουμε μέλη τύπου ακεραίου, αλλά με συγκεκριμένο πλήθος bits το καθένα.
- Παράδειγμα:

```
struct {
  unsigned int mode  : 2;
  unsigned int bool  : 1;
  unsigned int octal : 3;
} flags;
```

Διαχείριση συνδεδεμένων λιστών

```
/* File: listmanagement.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct listnode *Listptr;

struct listnode {
  int value;
  Listptr next;
};

int empty(Listptr);
int in(Listptr, int);
int n_th(Listptr, int, int *);
void insert_at_start(Listptr *, int);
void insert_at_end(Listptr *, int);
int delete(Listptr *, int);
void print(Listptr);

int main(void)
{ Listptr alist;
  int v;
  alist = NULL;                                        /* List is NULL */
                                              /* Check if list is empty */
  printf("List is%s empty\n", empty(alist) ? "" : " not");
  insert_at_start(&alist, 44);                 /* List is 44--> NULL */
  printf("List is "); print(alist);
  insert_at_end(&alist, 55);             /* List is 44--> 55--> NULL */
  printf("List is "); print(alist);
  insert_at_start(&alist, 33);     /* List is 33--> 44-> 55--> NULL */
  printf("List is "); print(alist);
  insert_at_end(&alist, 66); /* List is 33--> 44-> 55--> 66--> NULL */
  printf("List is "); print(alist);
                                              /* Check if list is empty */
  printf("List is%s empty\n", empty(alist) ? "" : " not");
                                                  /* Check membership */
  printf("55 is%s in list\n", in(alist, 55) ? "" : " not");
  printf("77 is%s in list\n", in(alist, 77) ? "" : " not");
```

```
  if (n_th(alist, 2, &v))                      /* Return 2nd element */
    printf("Item no 2 is %d\n", v);
  else
    printf("Item no 2 does not exist\n");
  if (n_th(alist, 6, &v))                      /* Return 6th element */
    printf("Item no 6 is %d\n", v);
  else
    printf("Item no 6 does not exist\n");
  printf("Deleting 55. %s\n", delete(&alist, 55) ? "OK!" : "Failed!");
                                  /* List is 33--> 44--> 66--> NULL */
  printf("List is "); print(alist);
  printf("Deleting 22. %s\n", delete(&alist, 22) ? "OK!" : "Failed!");
                                  /* List is 33--> 44--> 66--> NULL */
  printf("List is "); print(alist);
  printf("Deleting 33. %s\n", delete(&alist, 33) ? "OK!" : "Failed!");
                                       /* List is 44--> 66--> NULL */
  printf("List is "); print(alist);
  return 0;
}

int empty(Listptr list)                    /* Check if list is empty */
{ if (list == NULL)                           /* Is it really empty? */
    return 1;                                          /* Yes, it is */
  else
    return 0;                                        /* No, it isn't */
}

int in(Listptr list, int v)          /* Check if v is member of list */
{ while (list != NULL)          /* Visit list elements up to the end */
    if (list->value == v)   /* Did we find what we are looking for? */
      return 1;                                       /* Yes, we did */
    else
      list = list->next;                  /* No, go to next element */
  return 0;                             /* Finally, v is not in list */
}
```

```
int n_th(Listptr list, int n, int *vaddr)
           /* Return n-th element of list, if it exists, into vaddr */
{ while (list != NULL)    /* Maybe search up to the end of the list */
    if (n-- == 1) {              /* Did we reach the right element? */
      *vaddr = list->value;                        /* Yes, return it */
      return 1;                                       /* We found it */
    }
    else
      list = list->next;                  /* No, go to next element */
  return 0;                             /* Sorry, list is too short */
}

void insert_at_start(Listptr *ptraddr, int v)
                      /* Insert v as first element of list *ptraddr */
{ Listptr templist;
  templist = *ptraddr;                /* Save current start of list */
  *ptraddr = malloc(sizeof(struct listnode)); /* Space for new node */
  (*ptraddr)->value = v;                                 /* Put value */
  (*ptraddr)->next = templist;      /* Next element is former first */
}

void insert_at_end(Listptr *ptraddr, int v)
                       /* Insert v as last element of list *ptraddr */
{ while (*ptraddr != NULL)                      /* Go to end of list */
    ptraddr = &((*ptraddr)->next);/* Prepare what we need to change */
  *ptraddr = malloc(sizeof(struct listnode)); /* Space for new node */
  (*ptraddr)->value = v;                                 /* Put value */
  (*ptraddr)->next = NULL;              /* There is no next element */
}
```

```
int delete(Listptr *ptraddr, int v)
               /* Delete element v from list *ptraddr, if it exists */
{ Listptr templist;
  while ((*ptraddr) != NULL)   /* Visit list elements up to the end */
    if ((*ptraddr)->value == v) {    /* Did we find what to delete? */
      templist = *ptraddr;         /* Yes, save address of its node */
      *ptraddr = (*ptraddr)->next;        /* Bypass deleted element */
      free(templist);     /* Free memory for the corresponding node */
      return 1;                           /* We deleted the element */
    }
    else
      ptraddr = &((*ptraddr)->next);/* Prepare what we might change */
  return 0;        /* We did't find the element we were looking for */
}

void print(Listptr list)                  /* Print elements of list */
{ while (list != NULL) {       /* Visit list elements up to the end */
    printf("%d--> ", list->value);         /* Print current element */
    list = list->next;                        /* Go to next element */
  }
  printf("NULL\n");                            /* Print end of list */
}
```

```
% gcc -o listmanagement listmanagement.c
% ./listmanagement
List is empty
List is 44--> NULL
List is 44--> 55--> NULL
List is 33--> 44--> 55--> NULL
List is 33--> 44--> 55--> 66--> NULL
List is not empty
55 is in list
77 is not in list
Item no 2 is 44
Item no 6 does not exist
Deleting 55. OK!
List is 33--> 44--> 66--> NULL
Deleting 22. Failed!
List is 33--> 44--> 66--> NULL
Deleting 33. OK!
List is 44--> 66--> NULL
%
```

Διαχείριση δυαδικών δέντρων

```
/* File: treemanagement.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct tnode *Treeptr;

typedef struct tnode {
  char *word;
  Treeptr left;
  Treeptr right;
} Treenode;

Treeptr addtree(Treeptr, char *);
void treeprint(Treeptr, int);
void nodesprint(Treeptr);
int treedepth(Treeptr);
int treesearch(Treeptr, char *);

int main(int argc, char *argv[])
{ Treeptr p;
  char buf[80];
  p = NULL;                                  /* Initialize binary tree */
  while (scanf("%s", buf) != EOF)          /* Read words from input */
    p = addtree(p, buf);           /* and insert them into the tree */
  printf("Tree is:\n"),
  treeprint(p, 0);                  /* Kind of tree pretty printing */
  printf("\nNodes are:\n");
  nodesprint(p);          /* Print tree nodes in alphabetical order */
  printf("\n\nTree depth is %d\n", treedepth(p));
                                  /* Compute and print depth of tree */
  printf("\n");
  while (--argc) {                              /* For each argument */
    argv++;       /* check whether it coincides with any tree node */
    printf("%s found %s\n",
           (treesearch(p, *argv)) ? "   " : "not", *argv);
  }
  return 0;
}
```

```
Treeptr addtree(Treeptr p, char *w)   /* Insert word w into tree p */
{ int cond;
  if (p == NULL) {                              /* If tree is empty */
    p = malloc(sizeof(Treenode));   /* Allocate space for new node */
                                    /* Allocate space to copy word */
    p->word = malloc((strlen(w)+1) * sizeof(char));
    strcpy(p->word, w);                /* Copy word w to tree node */
    p->left = NULL;           /* Left subtree of new node is empty */
    p->right = NULL;         /* Right subtree of new node is empty */
  }
  else if ((cond = strcmp(w, p->word)) < 0)
                       /* Does word w precede word of current node? */
    p->left = addtree(p->left, w);
                            /* If yes, insert it into left subtree */
  else if (cond > 0)       /* Does it follow word of current node? */
    p->right = addtree(p->right, w);
                           /* If yes, insert it into right subtree */
  /* If it is the same with word of current node, do not insert it */
  return p;                                          /* Return tree */
}

void treeprint(Treeptr p, int indent)         /* Pretty print tree */
{ int i;
  if (p != NULL) {                          /* If tree is not empty */
    treeprint(p->right, indent+4);
                /* Print right subtree 4 places right of root node */
    for (i=0 ; i < indent ; i++)
      printf(" ");                      /* Take care for indentation */
    printf("%s\n", p->word);                     /* Print root node */
    treeprint(p->left, indent+4);
                 /* Print left subtree 4 places right of root node */
  }
}

void nodesprint(Treeptr p)                      /* Print tree nodes */
{ if (p != NULL) {                           /* If tree is not empty */
    nodesprint(p->left);                       /* Print left subtree */
    printf("%s ", p->word);                       /* Print root node */
    nodesprint(p->right);                    /* Print right subtree */
  }
}
```

```
int treedepth(Treeptr p)                /* Compute depth of tree p */
{ int n1, n2;
  if (p == NULL)                       /* Depth of empty tree is 0 */
    return 0;
  n1 = treedepth(p->left);        /* Compute depth of left subtree */
  n2 = treedepth(p->right);      /* Compute depth of right subtree */
  return (n1 > n2) ? n1+1 : n2+1;
     /* Return maximun of depths of left and right subtrees plus 1 */
}

int treesearch(Treeptr p, char *w)
                              /* Check whether word w is in tree p */
{ int cond;
  if (p == NULL)                                /* If tree is empty */
    return 0;                                 /* We didn't find word */
  if ((cond = strcmp(w, p->word)) == 0)
                  /* Word w is the same with word of current node */
    return 1;
  else if (cond < 0)         /* If w precedes word of current node */
    return treesearch(p->left, w);          /* Search left suftree */
  else                                                /* Otherwise */
    return treesearch(p->right, w);        /* search right subtree */
}
```

```
% gcc -o treemanagement treemanagement.c
% cat words.txt
some
words
to
insert
into
a
binary
tree
% ./treemanagement into found words < words.txt
Tree is:
    words
             tree
        to
some
        into
    insert
             binary
        a

Nodes are:
a binary insert into some to tree words

Tree depth is 4

    found into
not found found
    found words
%
```

Είσοδος και έξοδος

- Κάθε πρόγραμμα πρέπει να είναι σε θέση να επικοινωνεί με το περιβάλλον του, δηλαδή να διαβάζει δεδομένα και να γράφει αποτελέσματα, όποτε το χρειάζεται.
- Στην C, οι μηχανισμοί εισόδου/εξόδου παρέχονται από μία σειρά συναρτήσεων, οι οποίες περιλαμβάνονται στην πρότυπη βιβλιοθήκη εισόδου/εξόδου της γλώσσας. Όταν χρησιμοποιούμε συναρτήσεις της βιβλιοθήκης αυτής, πρέπει να συμπεριλαμβάνουμε στα προγράμματά μας και το σχετικό αρχείο επικεφαλίδας:

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

- Οι μονάδες εισόδου/εξόδου είναι τα ρεύματα. Μπορούμε να συσχετίσουμε ένα αρχείο, ανοίγοντάς το, με ένα ρεύμα, αλλά υπάρχουν και προκαθορισμένα ρεύματα, αυτό της πρότυπης εισόδου (`stdin`), της πρότυπης εξόδου (`stdout`) και της πρότυπης εξόδου για διαγνωστικά μηνύματα (`stderr`).
- Κάθε πρόγραμμα έχει, κατ΄ αρχήν, αντιστοιχίσει στο ρεύμα `stdin` το πληκτρολόγιο και στα ρεύματα `stdout` και `stderr` την οθόνη.
- Η διαφορά μεταξύ `stdout` και `stderr` συνίσταται στο ότι συνηθίζεται τα προγράμματα να εκτυπώνουν τα αποτελέσματά τους στο `stdout` και διαγνωστικά μηνύματα στο `stderr`.

- Σε πολλά περιβάλλοντα (Unix, Command Prompt των Windows, Cygwin, κλπ.) υπάρχει η δυνατότητα ένα πρόγραμμα που διαβάζει τα δεδομένα του από την πρότυπη είσοδο (`stdin`) να μπορεί να κληθεί με ανακατεύθυνση της πρότυπης εισόδου σε κάποιο αρχείο, οπότε θα διαβάζει πλέον τα δεδομένα του από το αρχείο αυτό. Παράδειγμα:

  ```
  % ./myprog < inp_file
  ```

  Εδώ, το πρόγραμμα `./myprog`, αντί να διαβάσει τα δεδομένα του από το πληκτρολόγιο, θα τροφοδοτηθεί με αυτά από το αρχείο `inp_file`.

- Ομοίως, μπορούμε να έχουμε και ανακατεύθυνση της πρότυπης εξόδου (`stdout`) σε αρχείο, οπότε, αντί να εκτυπώνονται τα αποτελέσματα ενός προγράμματος στην οθόνη, φυλάσσονται στο αρχείο αυτό. Παράδειγμα:

  ```
  % ./otherprog arg1 > out_file
  ```

  Εδώ, αντί να εκτυπωθούν τα αποτελέσματα του προγράμματος `./otherprog` στην οθόνη, θα φυλαχθούν στο αρχείο `out_file`.

- Επίσης, μπορούμε να δημιουργήσουμε σωληνώσεις από προγράμματα, δηλαδή η έξοδος του πρώτου προγράμματος της σωλήνωσης να είναι είσοδος για το δεύτερο πρόγραμμα, του οποίου η έξοδος να είναι είσοδος για το τρίτο, κ.ο.κ. Παράδειγμα:

  ```
  % ./prog1 22 13 | ./prog2 | ./prog3 27 | ./prog4 10 99
  ```

  Εδώ, τα προγράμματα `./prog1`, `./prog2`, `./prog3` και `./prog4` συμμετέχουν σε μία σωλήνωση, όπου η έξοδος καθενός είναι είσοδος στο επόμενο πρόγραμμα στη σωλήνωση.

- Επισημαίνεται ότι τις ανακατευθύνσεις (εισόδου και εξόδου) και τις σωληνώσεις τις διαχειρίζεται το περιβάλλον εκτέλεσης των προγραμμάτων (λειτουργικό σύστημα ή οτιδήποτε άλλο). Τα προγράμματα που συμμετέχουν σ΄ αυτές ποτέ δεν το μαθαίνουν. Με άλλα λόγια, οι χαρακτήρες `<`, `>` και `|`, καθώς και τα ονόματα αρχείων στις ανακατευθύνσεις **δεν είναι ορίσματα στη γραμμή εντολών** και δεν μπορούμε (και δεν χρειάζεται) να τα προσπελάσουμε μέσω `argc` και `argv`.

- Στη συνέχεια, περιγράφονται συνοπτικά οι πιο συχνά χρησιμοποιούμενες συναρτήσεις της πρότυπης βιβλιοθήκης εισόδου/εξόδου

- Περί εισόδου/εξόδου στην C, υπάρχει εκτεταμένη αναφορά στο Κεφάλαιο 7 του [KR] (σελ. 211–233).

- `int getchar(void)`
  Επιστρέφει τον επόμενο χαρακτήρα από το `stdin`, η `EOF`, αν έχει διαβαστεί όλη η είσοδος.

- `int putchar(int ch)`
  Γράφει τον χαρακτήρα `ch` στο `stdout`. Επιστρέφει τον χαρακτήρα αυτό, ή `EOF` σε περίπτωση λάθους.

- `int puts(const char *s)`
  Γράφει τη συμβολοσειρά `s` στο `stdout` και αμέσως μετά μία αλλαγή γραμμής. Επιστρέφει μη αρνητική τιμή σε επιτυχία, ή `EOF` σε περίπτωση λάθους. [α΄]

- `int printf(const char *format, <op>`$_1$`, <op>`$_2$`, ...)`
  Γράφει στο `stdout` τα ορίσματα `<op>`$_1$, `<op>`$_2$, ..., σύμφωνα με την οδηγία φόρμας `format`, και επιστρέφει το πλήθος των χαρακτήρων που γράφτηκαν.
  - Η συμβολοσειρά `format` περιλαμβάνει χαρακτήρες που μεταφέρονται αυτούσιοι στην έξοδο και προδιαγραφές μετατροπής για την εκτύπωση των `<op>`$_1$, `<op>`$_2$, ...
  - Μία προδιαγραφή μετατροπής αρχίζει με τον χαρακτήρα `%` και τερματίζει με ένα χαρακτήρα που δείχνει το είδος της μετατροπής που πρέπει να γίνει.

[α΄] Υπάρχει και η "αδελφή" συνάρτηση `char *gets(char *s)`, για την ανάγνωση από το `stdin` μίας συμβολοσειράς, αλλά δεν πρέπει να χρησιμοποιείται, για λόγους ασφαλείας. Μάλιστα, ο `gcc` δίνει σχετική προειδοποίηση όταν χρησιμοποιούμε αυτήν τη συνάρτηση σ΄ ένα πρόγραμμα. Αντ΄ αυτής, θα δούμε στη συνέχεια την ασφαλή συνάρτηση `fgets`.

– Η γενική μορφή μίας προδιαγραφής μετατροπής είναι η εξής:

$$\%[\tau\rho][\varepsilon\pi\pi][.\,\alpha\kappa\rho][\mu\mu]\chi\mu$$

Ό,τι βρίσκεται μέσα σε [και], μπορεί και να μην υπάρχει.

– Μεταξύ του % και του χαρακτήρα μετατροπής ($\chi\mu$) μπορεί να υπάρχουν:

* Κάποιοι τροποποιητές ($\tau\rho$), με οποιαδήποτε σειρά, δηλαδή ένα −, που επιβάλλει αριστερή στοίχιση, ή/και ένα +, που καθορίζει ότι ο αριθμός θα εμφανίζεται πάντα με πρόσημο, ή/και ένα 0, που, για αριθμούς, επιβάλλει τη συμπλήρωση του πλάτους πεδίου με μηδενικά στην αρχή, εφ΄ όσον έχουμε δεξιά στοίχιση, δηλαδή αν δεν έχει δοθεί και το − σαν τροποποιητής.
* Το ελάχιστο πλάτος πεδίου ($\varepsilon\pi\pi$). Συμπληρώνονται κενά αριστερά ή δεξιά, ανάλογα με τη στοίχιση, ή μηδενικά στην αρχή αν έχουμε δεξιά στοίχιση και έχει δοθεί και ή ένδειξη 0 ότι θέλουμε συμπλήρωση με μηδενικά.
* Μία τελεία . και ένας αριθμός, η ακρίβεια ($\alpha\kappa\rho$), που καθορίζει το μέγιστο αριθμό χαρακτήρων που θα εκτυπωθούν για ένα αλφαριθμητικό, τον αριθμό των ψηφίων[α΄] για αριθμό κινητής υποδιαστολής[β΄] ή τον ελάχιστο αριθμό ψηφίων για ακέραιο.
* Το μέγεθος μεταβλητής ($\mu\mu$), δηλαδή `h`, `l` ή `L`, αν πρόκειται για εκτύπωση `short int`, `long int` ή `long double`, αντίστοιχα.

[α΄] δεκαδικών για μετατροπές `f`, `e` ή `E` ή σημαντικών για μετατροπές `g` ή `G`

[β΄] αν δεν δοθεί ακρίβεια, αυτή θεωρείται ίση με 6

– Πιθανοί χαρακτήρες μετατροπής είναι:
  * d: Για αριθμούς στο δεκαδικό σύστημα.
  * o: Για απροσήμαστους οκταδικούς αριθμούς.
  * x, X: Για απροσήμαστους δεκαεξαδικούς αριθμούς. Με μετατροπή x, τα "ψηφία" μετά το 9 εμφανίζονται πεζά (a, b, c, d, e, f), ενώ με μετατροπή X, εμφανίζονται κεφαλαία (A, B, C, D, E, F).
  * u: Για απροσήμαστους αριθμούς στο δεκαδικό σύστημα.
  * c: Για χαρακτήρες.
  * s: Για συμβολοσειρές.
  * f: Για αριθμούς κινητής υποδιαστολής, σε αναπαράσταση χωρίς εκθέτη.
  * e, E: Για αριθμούς κινητής υποδιαστολής, σε αναπαράσταση με εκθέτη. Το σύμβολο της ύψωσης σε δύναμη του 10 εμφανίζεται σαν e ή E, ανάλογα με τον χαρακτήρα μετατροπής που χρησιμοποιήθηκε.
  * g, G: Για αριθμούς κινητής υποδιαστολής, σε αναπαράσταση με ή χωρίς εκθέτη, ανάλογα με το μέγεθος του εκθέτη σε σχέση με την ακρίβεια που έχει ζητηθεί. Αν ο εκθέτης είναι μικρότερος του -4 ή μεγαλύτερος από ή ίσος με την ακρίβεια, εφαρμόζεται μετατροπή e ή E (ανάλογα με το αν έχουμε g ή G), αλλιώς εφαρμόζεται μετατροπή f. Τα μηδενικά και η υποδιαστολή στο τέλος δεν εμφανίζονται.
  * %: Για εκτύπωση του %.

– Παραδείγματα εκτυπώσεων:

`int x = 3456`		`double d = -3768.08549`	
Προδιαγραφή	Εκτύπωση	Προδιαγραφή	Εκτύπωση
`%d`	`3456`	`%f`	`-3768.085490`
`%7d`	`␣␣␣3456`	`%14.4f`	`␣␣␣␣-3768.0855`
`%2d`	`3456`	`%-14.3f`	`-3768.085␣␣␣␣␣`
`%-7d`	`3456␣␣␣`	`%e`	`-3.768085e+03`
`%+7d`	`␣␣+3456`	`%14.4E`	`␣␣␣-3.7681E+03`
`%07d`	`0003456`	`%-14.3e`	`-3.768e+03␣␣␣␣`
`%7.5d`	`␣␣03456`	`%g`	`-3768.09`
`%7o`	`␣␣␣6600`	`%14.4g`	`␣␣␣␣␣␣␣␣-3768`
`%-7x`	`d80␣␣␣␣`	`%-14.3G`	`-3.77E+03␣␣␣␣␣`
`%7.6X`	`␣000D80`	`%014.4f`	`-00003768.0855`

`char *p = "this is a test phrase"`	
Προδιαγραφή	Εκτύπωση
`%s`	`this is a test phrase`
`%25s`	`␣␣␣␣this is a test phrase`
`%12s`	`this is a test phrase`
`%.12s`	`this is a te`
`%-25s`	`this is a test phrase␣␣␣␣`
`%25.12s`	`␣␣␣␣␣␣␣␣␣␣␣␣this is a te`
`%-25.12s`	`this is a te␣␣␣␣␣␣␣␣␣␣␣␣`

- `int scanf(const char *format, <op>`$_1$`, <op>`$_2$`, ...)`
  Διαβάζει από το `stdin` δεδομένα, σύμφωνα με την οδηγία φόρμας `format`, και τα αποθηκεύει εκεί που δείχνουν οι **δείκτες** `<op>`$_1$, `<op>`$_2$, ... Επιστρέφει το πλήθος των δεδομένων που διαβάστηκαν ή `EOF`, αν έχει διαβαστεί όλη η είσοδος.
  - Ένα δεδομένο εισόδου είναι μία ακολουθία από μη λευκούς χαρακτήρες που τερματίζει στον επόμενο λευκό χαρακτήρα,[α΄] ή στον επόμενο χαρακτήρα που αναμένεται να διαβαστεί με βάση το `format`, ή όταν εξαντληθεί το πλάτος πεδίου.
  - Η συμβολοσειρά `format` μπορεί να περιλαμβάνει:
    * Κενά ή στηλογνώμονες, που επιβάλλουν την κατανάλωση όλων των λευκών χαρακτήρων από την τρέχουσα θέση της εισόδου και μετά.
    * Άλλους χαρακτήρες, εκτός από τον %, που πρέπει να συμφωνούν με τον επόμενο μη λευκό χαρακτήρα στην είσοδο, αλλιώς η `scanf` σταματά το διάβασμα.

[α΄] κενό, στηλογνώμονας ή αλλαγή γραμμής

* Προδιαγραφές μετατροπής, που σχηματίζονται, με αυτή τη σειρά, από:
  - τον χαρακτήρα %
  - ένα προαιρετικό χαρακτήρα *, που δείχνει ότι το δεδομένο που διαβάστηκε δεν πρέπει να φυλαχθεί
  - ένα προαιρετικό αριθμό που καθορίζει το μέγιστο πλάτος πεδίου
  - ένα προαιρετικό `h`, για την ανάγνωση `short int`, ή ένα `l`, για `long int` ή `double`, ή ένα `L` για `long double`.
  - ένα χαρακτήρα μετατροπής, αντίστοιχα με αυτούς της `printf` (`d`, `o`, `x`, `u`, `c`, `s`, `f`, `e`, `g`, `%`) ή
    `[...]`, το οποίο προκαλεί την ανάγνωση της μεγαλύτερης ακολουθίας χαρακτήρων στην είσοδο, από αυτούς που περιλαμβάνονται μέσα στα `[` και `]`, ή
    `[^...]`, το οποίο είναι παρόμοιο με το προηγούμενο, αλλά διαβάζονται χαρακτήρες που **δεν** περιέχονται μέσα στα `[` και `]`

– Επισημαίνεται ότι η χρήση της `scanf` για ανάγνωση συμβολοσειρών παρουσιάζει παρόμοιο πρόβλημα ασφάλειας με αυτό της συνάρτησης `gets`, το οποίο όμως αντιμετωπίζεται αν δηλωθεί και μέγιστο πλάτος πεδίου (π.χ. `%20s`) για την ανάγνωση.

- `FILE *fopen(const char *filename,`
  `const char *mode)`

  Ανοίγει το αρχείο με όνομα `filename` για να το χρησιμοποιήσουμε με τον τρόπο που περιγράφεται στο `mode`. Επιστρέφει ένα ρεύμα (ή δείκτη σε αρχείο), με βάση το οποίο μπορούμε στη συνέχεια να αναφερόμαστε στο αρχείο, ή `NULL`, αν για κάποιο λόγο δεν ήταν δυνατόν να ανοίξει το αρχείο.
  - Το `FILE` είναι μία δομή (`typedef`'ed από το `stdio.h`) με κατάλληλα μέλη για να γίνεται ο χειρισμός του αρχείου.
  - Το `mode` μπορεί να είναι:
    * `"r"`: Για διάβασμα από υπάρχον αρχείο.
    * `"w"`: Για γράψιμο σε αρχείο. Αν το αρχείο δεν υπάρχει, δημιουργείται. Αν υπάρχει, τα προηγούμενα περιεχόμενά του διαγράφονται και το γράψιμο αρχίζει από την αρχή του αρχείου.
    * `"a"`: Για γράψιμο σε αρχείο με προσάρτηση στο τέλος του των νέων δεδομένων, αν το αρχείο υπάρχει ήδη, χωρίς διαγραφή των προηγούμενων περιεχομένων του.
    * `"r+"`: Για διάβασμα και γράψιμο, οπουδήποτε μέσα σε υπάρχον αρχείο, χωρίς διαγραφή των προηγούμενων περιεχομένων του.
    * `"w+"`: Για διάβασμα και γράψιμο, οπουδήποτε μέσα στο αρχείο, με διαγραφή των προηγούμενων περιεχομένων του, εφ' όσον αυτό υπάρχει ήδη.
    * `"a+"`: Για διάβασμα από οπουδήποτε μέσα από το αρχείο και γράψιμο μόνο στο τέλος του, χωρίς διαγραφή των προηγούμενων περιεχομένων του.

  - Σε ορισμένα συστήματα, γίνεται διάκριση μεταξύ αρχείων κειμένου και δυαδικών αρχείων (στο Unix πάντως, όχι), οπότε εκεί, αν θέλουμε να χειριστούμε ένα δυαδικό αρχείο, πρέπει στο `mode` να προστεθεί και ο χαρακτήρας `b`, δηλαδή να έχουμε, ανάλογα με την περίπτωση `"rb"`, `"wb"`, `"ab"`, `"r+b"` (`"rb+"`), `"w+b"` (`"wb+"`), ή `"a+b"` (`"ab+"`).

- `int fclose(FILE *fp)`
  Κλείνει το ρεύμα `fp`. Επιστρέφει `0` ή `EOF`, σε περίπτωση επιτυχίας ή αποτυχίας, αντίστοιχα.

- `char *fgets(char *buf, int max, FILE *fp)`
  Διαβάζει το πολύ `max-1` χαρακτήρες από το ρεύμα `fp`, μέχρι την αλλαγή γραμμής (`\n`). Οι χαρακτήρες που διαβάστηκαν (μαζί και με το `\n`) φυλάσσονται στη συμβολοσειρά `buf`, η οποία τερματίζεται κανονικά με `\0`. Επιστρέφει το `buf`, ή `NULL` αν έχει διαβαστεί όλη η είσοδος. [α΄]

- `int feof(FILE *fp)`
  Επιστρέφει μη μηδενική τιμή αν έχουν διαβαστεί όλα τα δεδομένα από το ρεύμα `fp`, δηλαδή έχουμε φτάσει στο τέλος του αρχείου που αντιστοιχεί στο ρεύμα, ή, αλλιώς, `0`.

[α΄] Επισήμως, υπάρχει και η συνάρτηση `char *gets(char *buf)`, η οποία κάνει περίπου ό,τι και η `fgets`. Διαβάζει από το ρεύμα `stdin`, χωρίς να βάζει κάποιο πάνω όριο στο πλήθος των χαρακτήρων που θα διαβαστούν και, γι΄ αυτό ακριβώς, για λόγους ασφαλείας, απαγορεύεται η χρήση της. Επίσης, η `gets` δεν τοποθετεί τον χαρακτήρα αλλαγής γραμμής (`\n`) που διάβασε μέσα στο `buf`. Αν θέλουμε να διαβάσουμε μία γραμμή χαρακτήρων από την πρότυπη είσοδο, μπορούμε να χρησιμοποιήσουμε την `fgets`, δίνοντας σαν τελευταίο όρισμα το `stdin`, αντί να χρησιμοποιήσουμε την `gets`.

- `int getc(FILE *fp)`
  Επιστρέφει τον επόμενο χαρακτήρα από το ρεύμα `fp`, ή `EOF`, αν έχουμε φτάσει στο τέλος του αντίστοιχου αρχείου.

- `int putc(int ch, FILE *fp)`
  Γράφει τον χαρακτήρα `ch` στο ρεύμα `fp`. Επιστρέφει τον χαρακτήρα αυτό, ή `EOF` σε περίπτωση λάθους.

- `int ungetc(int ch, FILE *fp)`
  Γυρίζει πίσω στο ρεύμα `fp` τον χαρακτήρα `ch`, έτσι ώστε να διαβαστεί πάλι σε επόμενη ανάγνωση. Επιστρέφει τον χαρακτήρα αυτό, ή `EOF` σε περίπτωση λάθους.

- `int fprintf(FILE *fp, const char *format,`
  `<op>`$_1$`, <op>`$_2$`, ...)`
  Ίδια με την `printf`, μόνο που γράφει στο ρεύμα `fp`, αντί για το `stdout`.

- `int fscanf(FILE *fp, const char *format,`
  `<op>`$_1$`, <op>`$_2$`, ...)`
  Ίδια με την `scanf`, μόνο που διαβάζει από το ρεύμα `fp`, αντί από το `stdin`.

- `int sprintf(char *str, const char *format,`
  `<op>`$_1$`, <op>`$_2$`, ...)`
  Ίδια με την `printf`, μόνο που γράφει στη συμβολοσειρά `str`, αντί για το ρεύμα `stdout`.

- `int sscanf(char *str, const char *format,`
  `<op>`$_1$`, <op>`$_2$`, ...)`
  Ίδια με την `scanf`, μόνο που διαβάζει από τη συμβολοσειρά `str`, αντί από το ρεύμα `stdin`.

- `size_t fread(void *ptr, size_t size, size_t count, FILE *fp)` [α′]

  Διαβάζει από το ρεύμα `fp` το πολύ `count` δεδομένα μεγέθους `size` το καθένα και τα τοποθετεί από τη διεύθυνση `ptr` και μετά. Επιστρέφει το πλήθος των δεδομένων που διαβάστηκαν. Ο έλεγχος τέλους της εισόδου μπορεί να γίνει με τη συνάρτηση `feof`.

- `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *fp)`

  Γράφει στο ρεύμα `fp` το πολύ `count` δεδομένα μεγέθους `size` το καθένα, παίρνοντάς τα από τη διεύθυνση `ptr` και μετά. Επιστρέφει το πλήθος των δεδομένων που γράφτηκαν. Αν γραφούν λιγότερα από `count` δεδομένα, αυτό θα οφείλεται σε κάποιο λάθος που συνέβη.

- `int fseek(FILE *fp, long offset, int origin)`

  Θέτει την τρέχουσα θέση στο ρεύμα `fp` να είναι:

  - `offset` (≥ 0) χαρακτήρες από την αρχή, αν το `origin` έχει τεθεί `SEEK_SET`.
  - `offset` χαρακτήρες από την τρέχουσα θέση, αν το `origin` έχει τεθεί `SEEK_CUR`.
  - `offset` χαρακτήρες από το τέλος, αν το `origin` έχει τεθεί `SEEK_END`.

  Επιστρέφει 0 σε περίπτωση επιτυχίας.

[α′] Ο τύπος `size_t` είναι ο `unsigned int`, `typedef`′ed.

- `long ftell(FILE *fp)`
  Επιστρέφει την τρέχουσα θέση στο ρεύμα `fp`, ή `-1L` σε περίπτωση λάθους.

- `int fflush(FILE *fp)`
  Γράφει στο ρεύμα εξόδου `fp` ό,τι πιθανώς βρίσκεται στην ενδιάμεση περιοχή αποθήκευσης. Επιστρέφει `0` ή `EOF`, σε περίπτωση επιτυχίας ή αποτυχίας, αντίστοιχα.

- `void perror(const char *s)`
  Εκτυπώνει τη συμβολοσειρά `s` και μία περιγραφή του πιο πρόσφατου λάθους που έχει συμβεί.
  - Η συνάρτηση `perror` είναι πολύ κατατοπιστική όταν χρησιμοποιούμε συναρτήσεις της πρότυπης βιβλιοθήκης της C, οι οποίες ενδέχεται να προκαλέσουν κάποιο λάθος, για παράδειγμα η `malloc` ή οι συναρτήσεις διαχείρισης αρχείων, όπως η `fopen`, και μας ενδιαφέρει να μάθουμε ποιο ακριβώς λάθος προκλήθηκε.
  - Σχετική με τη συνάρτηση `perror` είναι και μία εξωτερική μεταβλητή `errno`, η οποία έχει σαν τιμή έναν ακέραιο που αντιστοιχεί στο πιο πρόσφατο λάθος που έχει συμβεί.
  - Αν θέλουμε να χρησιμοποιήσουμε τη μεταβλητή `errno` μέσα σ΄ ένα πρόγραμμα, πρέπει να έχουμε συμπεριλάβει και το αρχείο επικεφαλίδας στο οποίο ορίζεται:

    ```
    #include <errno.h>
    ```

Μία επισήμανση που πρέπει να γίνει είναι ότι δεν μπορούμε να χρησιμοποιήσουμε οποιαδήποτε συνάρτηση σε οποιοδήποτε ρεύμα. Εξαρτάται από το `mode` που δόθηκε όταν άνοιξε το αντίστοιχο αρχείο.

Αντιγραφή αρχείων

```
/* File: filecopy.c */
#include <stdio.h>

int main(int argc, char *argv[])
{ FILE *ifp, *ofp;
  int n;
  char buf[1024];
  if (argc != 3) {
    fprintf(stderr,
            "Usage: %s <soure-file> <target-file>\n", argv[0]);
    return 1;
  }
  if ((ifp = fopen(argv[1], "rb")) == NULL) { /* Open source file */
    perror("fopen source-file");
    return 1;
  }
  if ((ofp = fopen(argv[2], "wb")) == NULL) { /* Open target file */
    perror("fopen target-file");
    return 1;
  }
  while (!feof(ifp)) {  /* While we don't reach the end of source */
              /* Read characters from source file to fill buffer */
    n = fread(buf, sizeof(char), sizeof(buf), ifp);
                          /* Write characters read to target file */
    fwrite(buf, sizeof(char), n, ofp);
  }
  fclose(ifp);
  fclose(ofp);
  return 0;
}
```

```
% gcc -o filecopy filecopy.c
% ./filecopy
Usage: ./filecopy <soure-file> <target-file>
% ./filecopy bla foo
fopen source-file: No such file or directory
%
% cat helloworld.c
/* File: helloworld.c */
#include <stdio.h>

main()
{ printf("Hello world\n");
}
% ./filecopy helloworld.c newhelloworld.c
% cat newhelloworld.c
/* File: helloworld.c */
#include <stdio.h>

main()
{ printf("Hello world\n");
}
% ls -l helloworld.c newhelloworld.c
-rw-------   1 ip       www           81 Dec 18 21:45 newhelloworld.c
-rw-r-----   1 ip       www           81 Oct 13 12:42 helloworld.c
% diff helloworld.c newhelloworld.c
%
% ./filecopy filecopy Copy\ of\ filecopy
% ls -l filecopy Copy\ of\ filecopy
-rw-------   1 ip       www         6884 Dec 18 21:46 Copy of filecopy
-rwx------   1 ip       www         6884 Dec 18 21:34 filecopy*
% cmp filecopy Copy\ of\ filecopy
%
```

Εκτύπωση γραμμών εισόδου με αντίστροφη σειρά

```
/* File: revinput.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Slistnode *SListptr;

struct Slistnode {                    /* A list node structure */
  char *line;                        /* with a string as member */
  SListptr next;
};

void Sinsert_at_start(SListptr *, char *);
void Sprint(SListptr);

int main(void)
{ char buf[1024];
  SListptr list;
  list = NULL;    /* Initialize list to store lines to be read */
  while (fgets(buf, sizeof buf, stdin) != NULL) /* Read a line */
                     /* and insert it at the start of the list */
    Sinsert_at_start(&list, buf);
  Sprint(list);     /* Print the list, i.e. the input reversed */
  return 0;
}

void Sinsert_at_start(SListptr *ptraddr, char *buf)
                    /* The well-known insert_at_start function */
{ SListptr templist;
  templist = *ptraddr;
  *ptraddr = malloc(sizeof(struct Slistnode));
  (*ptraddr)->line = malloc((strlen(buf)+1) * sizeof(char));
  strcpy((*ptraddr)->line, buf);
  (*ptraddr)->next = templist;
}

void Sprint(SListptr list)          /* Just print out the list */
{ while (list != NULL) {
    printf("%s", list->line);
    list = list->next; }
}
```

```
% gcc -o revinput revinput.c
% ./revinput
These are some lines
to test program revinput.
Ok, one more line.
These are my last words
before the minus signs
---------------------------
^D
---------------------------
before the minus signs
These are my last words
Ok, one more line.
to test program revinput.
These are some lines
%
% cat helloworld.c
/* File: helloworld.c */
#include <stdio.h>

main()
{ printf("Hello world\n");
}
%
% ./revinput < helloworld.c
}
{ printf("Hello world\n");
main()

#include <stdio.h>
/* File: helloworld.c */
```

Ο προεπεξεργαστής της C

- Ο προεπεξεργαστής της C είναι ένα αυτόνομο υποσύστημα του λογισμικού μεταγλώττισης, το οποίο καλείται αυτόματα πριν από την πραγματική διαδικασία της μεταγλώττισης. Ο σκοπός του προεπεξεργαστή είναι να μετασχηματίσει το πηγαίο αρχείο C που του δόθηκε σ΄ ένα άλλο πηγαίο αρχείο, σύμφωνα με μία σειρά από οδηγίες που απευθύνονται σ΄ αυτόν. Το αποτέλεσμα του προεπεξεργαστή είναι αυτό το οποίο θα υποστεί τη διαδικασία της μεταγλώττισης.
- Οι γραμμές του πηγαίου αρχείου που αρχίζουν από # είναι οδηγίες προς τον προεπεξεργαστή.
- Ακολουθεί μία συνοπτική περιγραφή των πιο συχνά χρησιμοποιούμενων οδηγιών του προεπεξεργαστή. Σχετικά πιο αναλυτική αναφορά γίνεται στην παράγραφο §4.11 του [KR] (σελ. 128–133).

#include

- Χρησιμοποιείται για την εισαγωγή των περιεχομένων αρχείων επικεφαλίδας στη θέση που βρίσκεται η οδηγία. Ένα αρχείο επικεφαλίδας περιέχει συνήθως δηλώσεις πρωτοτύπων συναρτήσεων, ορισμούς συμβολικών σταθερών και μακροεντολών (μέσω `#define`), συμπεριλήψεις άλλων αρχείων επικεφαλίδας, δηλώσεις μορφής τύπων δεδομένων (μέσω `struct`, `enum`, `union`, `typedef`), κλπ. Σε αρχεία επικεφαλίδας, **δεν** συνηθίζεται να έχουμε κώδικα προγράμματος. Επίσης, **δεν** κάνουμε `#include` πηγαία αρχεία κώδικα. Η οδηγία αυτή εμφανίζεται με δύο εκδοχές.

- Πρώτη εκδοχή: `#include <sysfile.h>`
  Ο προεπεξεργαστής θα ψάξει να βρει το αρχείο `sysfile.h` σε καταλόγους που έχει δηλώσει με κάποιο τρόπο ο χρήστης, για παράδειγμα με την επιλογή `-I` κατά την κλήση του `gcc`, ή στους καταλόγους που βρίσκονται τα προκαθορισμένα αρχεία επικεφαλίδας της γλώσσας, οι οποίοι μπορεί να διαφέρουν από εγκατάσταση σε εγκατάσταση.

- Δεύτερη εκδοχή: `#include "myfile.h"`
  Ο προεπεξεργαστής θα ψάξει κατ΄ αρχήν να βρει το αρχείο `myfile.h` στον κατάλογο που βρίσκεται το πηγαίο αρχείο, και αν δεν βρεθεί εκεί, θα το αναζητήσει σε καταλόγους όπως θα γινόταν αν η οδηγία είχε δοθεί με την πρώτης της εκδοχή.

- Φυσικά, είναι επιτρεπτές και δηλώσεις:

```
#include <sys/types.h>
#include "mydir/anotherdir/myfile.h"
```

#define

- Χρησιμοποιείται για τον ορισμό συμβολικών σταθερών ή μακροεντολών. Όπου βρει ο προεπεξεργαστής σαν λεκτικό σύμβολο (όχι μέσα σε " ή σε σχόλια) τη συμβολική σταθερά ή την παραμετρική μακροεντολή που ορίζεται με την οδηγία, την αντικαθιστά με το κείμενο που της έχει αντιστοιχηθεί.

- Παράδειγμα ορισμού συμβολικής σταθεράς:

```
#define YES 1
```

  Ο προεπεξεργαστής θα αντικαταστήσει το YES, όπου το βρει σαν λεκτικό σύμβολο μέσα στο πηγαίο αρχείο με το 1 (όχι όμως στο `printf("YES\n")`, ούτε και στο `x = YESMAN++`).

- Παράδειγμα ορισμού μακροεντολής:

```
#define max(A, B)     ((A) > (B) ? (A) : (B))
```

  Αν βρει ο προεπεξεργαστής στη συνέχεια στο αρχείο κάποια εντολή της μορφής

```
x = max(p+q, r+s);
```

  θα την αντικαταστήσει με την

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

  Παρατηρήστε, στο παράδειγμα αυτό, ότι τα A και B θα υπολογισθούν τελικά δύο φορές, αφού τόσες βρίσκονται στο κείμενο αντικατάστασης, πράγμα που μπορεί στο συγκεκριμένο παράδειγμα να μην ενοχλεί, αλλά σε άλλες περιπτώσεις χρειάζεται προσοχή. [α΄]

[α΄] Πιστεύετε ότι η έκφραση `max(i++, j++)` θα αντικατασταθεί σωστά με βάση τι είχε στο μυαλό του μάλλον ο προγραμματιστής που την έγραψε;

- Οι παρενθέσεις γύρω από τις παραμέτρους στο κείμενο αντικατάστασης μίας μακροεντολής ενδέχεται, σε κάποιες περιπτώσεις, να είναι κρίσιμες. Παράδειγμα:

```
#define square(X)     X * X
```

  Πώς θα αντικατασταθεί το `square(a+1)`; [α΄] Αυτό ήταν που θέλαμε; [β΄] Πώς θα έπρεπε να δηλωθεί η μακροεντολή; [γ΄]

- Αρκετές από τις συναρτήσεις της γλώσσας που χρησιμοποιούμε στα προγράμματά μας είναι ορισμένες σαν μακροεντολές μέσα σε αρχεία επικεφαλίδας και όχι με κώδικα στη βιβλιοθήκη της γλώσσας. Για παράδειγμα, οι γνωστές μας συναρτήσεις `getchar` και `putchar` ορίζονται μέσα στο αρχείο `stdio.h` ως εξής:

```
#define getchar()     getc(stdin)
#define putchar(x)    putc(x, stdout)
```

[α΄] `a+1 * a+1`

[β΄] Όχι, βέβαια.

[γ΄] `#define square(X)     ((X) * (X))`

#if, #else, #elif, #endif, #ifdef, #ifndef

- Στην C, υπάρχει η δυνατότητα μεταγλώττισης υπό συνθήκη. Ό,τι περιλαμβάνεται μεταξύ ενός #if και του αμέσως επόμενου #endif, που δεν έχει αντιστοιχηθεί με κάποιο άλλο #if, θα μεταγλωττιστεί μόνο αν η ακέραια παράσταση που βρίσκεται μετά το #if αποτιμηθεί σε μη μηδενική τιμή.
- Συνήθως, οι ακέραιες παραστάσεις μετά το #if συγκρίνουν τιμές συμβολικών σταθερών, που έχουν αντιστοιχηθεί σε ακέραιους, μεταξύ τους ή με ακέραιες σταθερές, μέσω των τελεστών σύγκρισης της C (==, !=, >, κλπ.).
- Με την οδηγία #else, αρχίζει το τμήμα μίας δομής #if/#endif που θα μεταγλωττιστεί αν η παράσταση μετά το #if είναι ψευδής (ίση με μηδέν).
- Η οδηγία #elif μπορεί να χρησιμοποιηθεί για πιο σύντομη γραφή μίας #else της οποίας το σώμα αποτελείται από μία #if, αποφεύγοντας έτσι και το επιπλέον #endif.
- Παράδειγμα:

```
#if SYSTEM == SYSV
   #define HDR "sysv.h"
#elif SYSTEM == BSD
   #define HDR "bsd.h"
#elif SYSTEM == MSDOS
   #define HDR "msdos.h"
#else
   #define HDR "default.h"
#endif
#include HDR
```

- Κάποια ειδική παράσταση που μπορεί να υπάρχει μετά από ένα `#if` είναι η `defined(NAME)`, που αποτιμάται σε `1` αν το `NAME` έχει οριστεί με `#define`, αλλιώς σε `0`. Η οδηγία

  ```
  #if defined(NAME)
  ```

  είναι ισοδύναμη με την

  ```
  #ifdef NAME
  ```

- Στις παραστάσεις μετά από ένα `#if` μπορούμε να έχουμε και άρνηση (`!`). Για παράδειγμα, αν θέλουμε να εξασφαλίσουμε ότι τα περιεχόμενα του αρχείου `hdr.h` θα συμπεριληφθούν μία μόνο φορά σε μία πολύπλοκη εφαρμογή με πολλές συμπεριλήψεις, θα μπορούσαμε να το δομήσουμε ως εξής:

  ```
  #if !defined(HDR)
  #define HDR
  /* The real contents of hdr.h go here */
  #endif
  ```

  Το `#if !defined(HDR)` θα μπορούσε να γραφεί ισοδύναμα και σαν `#ifndef HDR`

- Δομές `#if` (`#ifdef`, `#ifndef`)/`#endif` μπορεί να είναι εμφωλευμένες. Με τη βοήθειά τους μπορούμε να προσομοιώσουμε εμφωλευμένα σχόλια, κάτι που με τον συνήθη τρόπο γραφής σχολίων (`/* ... */`) δεν είναι δυνατόν. *Πώς;*

Ταξινόμηση πινάκων

- Υπάρχουν διάφορες μέθοδοι, άλλες λιγότερο, άλλες περισσότερο αποδοτικές, για την ταξινόμηση των στοιχείων ενός πίνακα.
- Η ταξινόμηση ενός πίνακα έχει νόημα όταν υπάρχει μία σχέση διάταξης στο σύνολο από το οποίο παίρνουν τιμές τα στοιχεία του πίνακα. Για αριθμούς (ακέραιους ή κινητής υποδιαστολής) έχουμε την αριθμητική διάταξη, για συμβολοσειρές την αλφαβητική. Για άλλους τύπους δεδομένων, πρέπει να έχει ορισθεί σαφώς η σχέση διάταξης. Για παράδειγμα, διάταξη μεταξύ δομών μπορεί να οριστεί με βάση τη διάταξη ως προς ένα μέλος-κλειδί της δομής (αριθμητικό ή συμβολοσειρά).
- Στη συνέχεια, οι αλγόριθμοι, τα παραδείγματα και τα προγράμματα που παρουσιάζονται στοχεύουν στην ταξινόμηση πινάκων, σε αύξουσα σειρά, με στοιχεία που είναι πραγματικοί αριθμοί διπλής ακρίβειας.
- Οι μέθοδοι ταξινόμησης μπορούν να προσαρμοσθούν εύκολα και για περιπτώσεις άλλων μονοδιάστατων δομών δεδομένων, για παράδειγμα συνδεδεμένων λιστών.

- Ένα θέμα που έχει γενικότερο ενδιαφέρον στους αλγορίθμους είναι ο χρόνος που χρειάζεται για την εκτέλεσή τους, ανάλογα με το μέγεθος της εισόδου τους. Αυτό εκφράζεται με την πολυπλοκότητα χρόνου τους. Οι μέθοδοι ταξινόμησης έχουν διάφορες πολυπλοκότητες χρόνου, οι οποίες αντανακλούν την αποδοτικότητά τους.

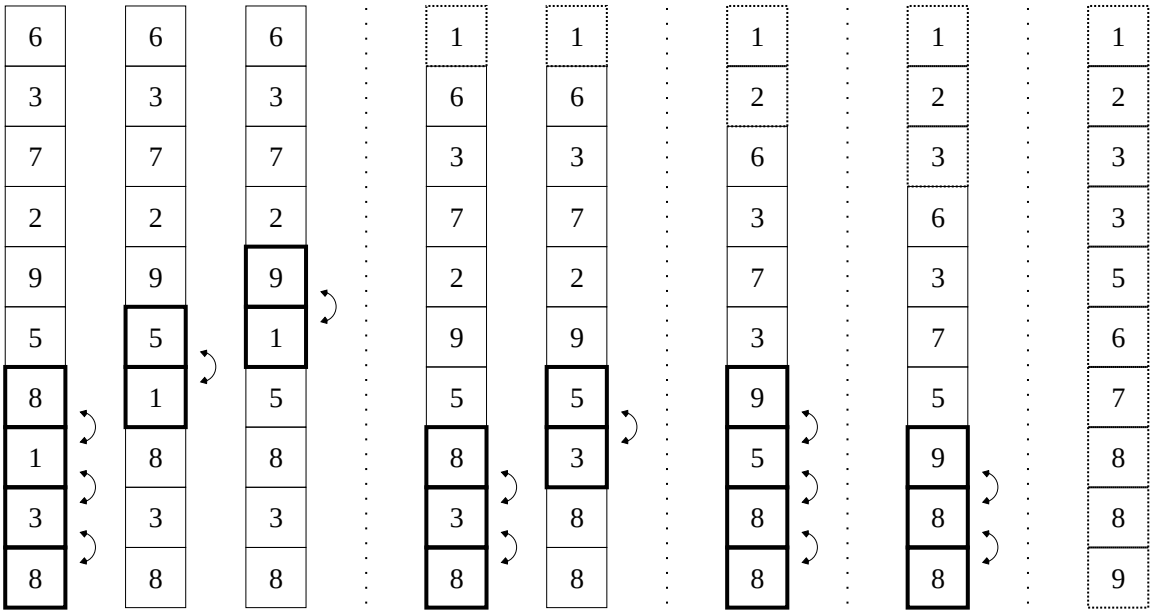
- Για να εκφράσουμε την πολυπλοκότητα ενός αλγορίθμου, συνήθως χρησιμοποιούμε τον συμβολισμό O. Όταν η είσοδος ενός αλγορίθμου έχει μέγεθος n (για παράδειγμα το πλήθος των στοιχείων ενός πίνακα που θέλουμε να ταξινομήσουμε), λέγοντας ότι η πολυπλοκότητα χρόνου του αλγορίθμου είναι $O(f(n))$, εννοούμε ότι ο χρόνος εκτέλεσής του, $T(n)$, δεν έχει μεγαλύτερο ρυθμό αύξησης από αυτόν της συνάρτησης $f(n)$, όσο αυξάνει το n. Δηλαδή υπάρχουν C και n_0 τέτοια ώστε $T(n) < C \cdot f(n)$ για κάθε $n > n_0$.

- Για παράδειγμα, η πολυπλοκότητα του αλγορίθμου για την εύρεση της μέσης τιμής n αριθμών είναι $O(n)$, ενώ η πολυπλοκότητα του αλγορίθμου κατασκευής ενός μαγικού τετραγώνου $n \times n$ είναι $O(n^2)$.

- Σε ορισμένες περιπτώσεις, η χρονική απόδοση ενός αλγορίθμου για είσοδο μεγέθους n εξαρτάται και από το ποια είναι η συγκεκριμένη είσοδος. Στις περιπτώσεις αυτές, μπορούμε να αναφερόμαστε στην πολυπλοκότητα χείριστης περίπτωσης (για την πιο "δύσκολη" είσοδο) ή στη μέση πολυπλοκότητα (μέση τιμή απόδοσης για όλες τις πιθανές εισόδους).

- *Η μέθοδος της φυσαλίδας*

  Σύγκρινε ζευγάρια διαδοχικών στοιχείων, από κάτω προς τα επάνω, και όταν βρίσκεις δύο που δεν είναι στη σωστή σειρά, αντιμετάθεσέ τα. Μετά από το πρώτο πέρασμα, πρώτο στοιχείο θα είναι το μικρότερο όλων. Κάνε και δεύτερο πέρασμα, για τα στοιχεία από το δεύτερο και μετά, οπότε έτσι θα έλθει και το δεύτερο κατά σειρά στη θέση του. Μετά από $n-1$ περάσματα συνολικά, ο πίνακας θα έχει ταξινομηθεί.

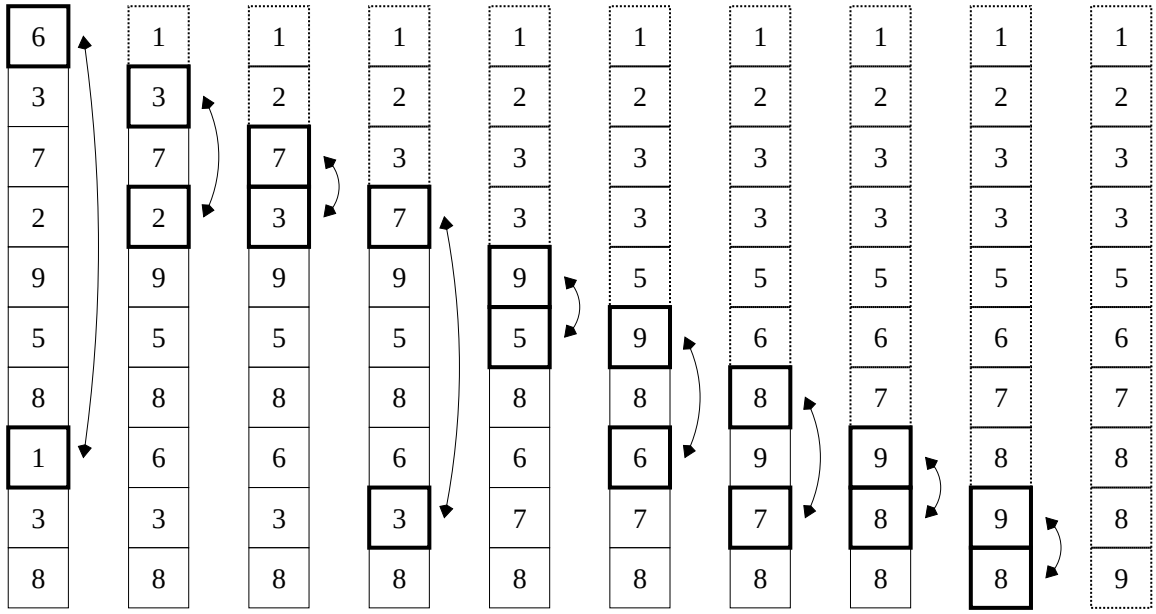


πολυπλοκότητα $O(n^2)$

Για $i = 1, 2, \ldots, n-1$
  Για $j = n-1, n-2, \ldots, i$
    Αν $x_{j-1} > x_j$ **τότε**
      Αντιμετάθεσε x_{j-1} και x_j

- *Η μέθοδος της επιλογής*

  Βρες το μικρότερο από όλα τα στοιχεία και αντιμετάθεσέ το με το πρώτο. Βρες το μικρότερο από τα υπόλοιπα και αντιμετάθεσέ το με το δεύτερο. Μετά από $n-1$ επιλογές και αντιμεταθέσεις συνολικά, ο πίνακας θα έχει ταξινομηθεί.

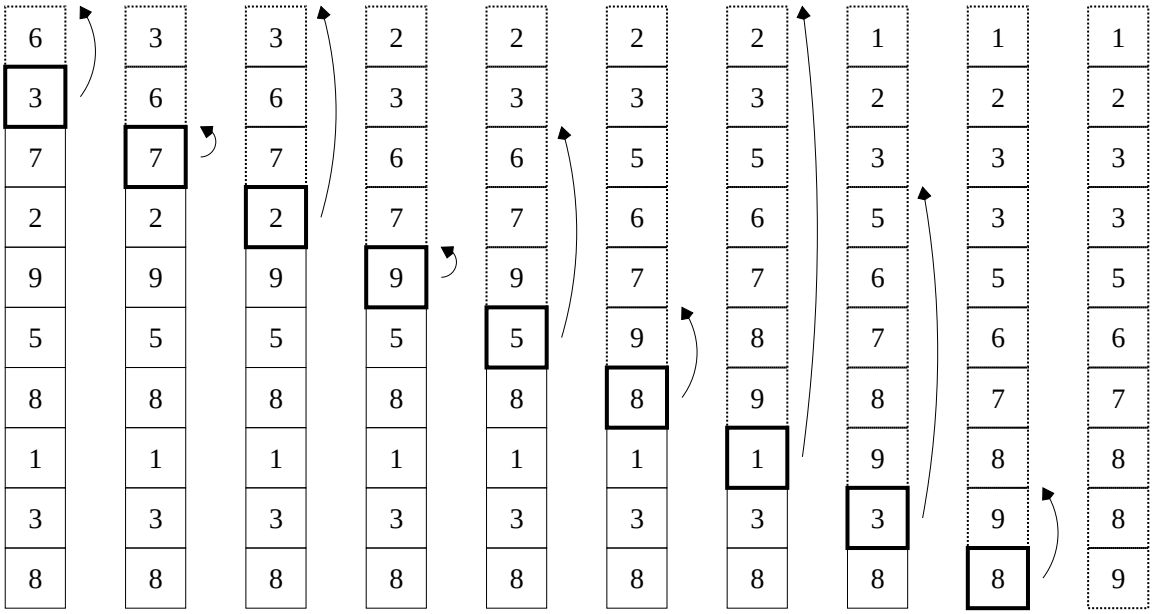


πολυπλοκότητα $O(n^2)$

Για $i = 1, 2, \ldots, n-1$
  $min = i-1$
  Για $j = i, i+1, \ldots, n-1$
    Αν $x_j < x_{min}$ **τότε**
      $min = j$
  Αντιμετάθεσε x_{i-1} και x_{min}

- *Η μέθοδος της εισαγωγής*

  Τοποθέτησε το δεύτερο στοιχείο πριν ή μετά το πρώτο ώστε να είναι στη σωστή σειρά. Τοποθέτησε το τρίτο στη σωστή θέση στα ήδη ταξινομημένα πρώτο και δεύτερο. Μετά τοποθέτησε το τέταρτο στα τρία πρώτα. Μετά από $n-1$ εισαγωγές συνολικά, ο πίνακας θα έχει ταξινομηθεί.

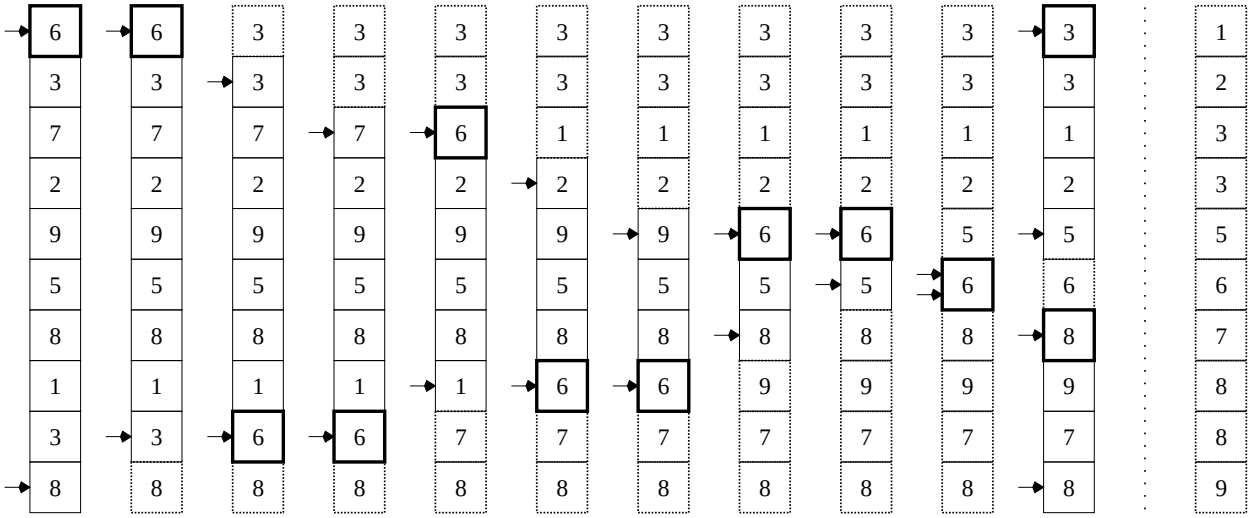


πολυπλοκότητα $O(n^2)$

Για $i = 1, 2, \ldots, n-1$
  $j = i-1$
  Ενόσω $j \geq 0$ **και** $x_j > x_{j+1}$
    Αντιμετάθεσε x_j και x_{j+1}
    $j = j-1$

- *Η γρήγορη μέθοδος*

  Με βάση το πρώτο στοιχείο σαν οδηγό, χώρισε τον πίνακα σε δύο άλλους, έναν που περιέχει στοιχεία μικρότερα ή ίσα με τον οδηγό και έναν με μεγαλύτερα ή ίσα. Ταξινόμησε τους δύο αυτούς πίνακες με την ίδια μέθοδο αναδρομικά (εφ΄ όσον έχουν τουλάχιστον δύο στοιχεία), οπότε το τελικό αποτέλεσμα είναι η παράθεση των ταξινομημένων αυτών πινάκων με ενδιάμεση παρεμβολή του οδηγού στοιχείου.



μέση πολυπλοκότητα $O(n \log n)$

Συνάρτηση $qs(x,\ up,\ down)$
  $start = up$
  $end = down$
  Ενόσω $up < down$
    Ενόσω $x_{down} \geq x_{up}$ **και** $up < down$
      $down = down{-}1$
    Αν $up \neq down$ **τότε**
      Αντιμετάθεσε x_{up} και x_{down}
      $up = up{+}1$
    Ενόσω $x_{up} \leq x_{down}$ **και** $up < down$
      $up = up{+}1$
    Αν $up \neq down$ **τότε**
      Αντιμετάθεσε x_{up} και x_{down}
      $down = down{-}1$
  Αν $start < up{-}1$ **τότε**
    Κάλεσε $qs(x,\ start,\ up{-}1)$
  Αν $end > down{+}1$ **τότε**
    Κάλεσε $qs(x,\ down{+}1,\ end)$

Για να ταξινομήσουμε τον πίνακα x που έχει n στοιχεία, θα πρέπει να καλέσουμε $qs(x,\ 0,\ n{-}1)$.

- Άλλες μέθοδοι ταξινόμησης
  - Η μέθοδος του σωρού, $O(n \log n)$
  - Η μέθοδος της συγχώνευσης, $O(n \log n)$
  - Η μέθοδος του Shell, $O(n^2)$ (σελ. 94 [KR])

Μέθοδοι ταξινόμησης

```
/* File: sorting.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void bubblesort(int, double *);
void selectsort(int, double *);
void insertsort(int, double *);
void quicksort(int, double *);
void quicksort_body(double *, int, int);
void swapd(double *, double *);

int main(int argc, char *argv[])
{ char method = 'b', *name; /* Default sorting method is bubblesort */
  int i, n = 10;                        /* Default array size is 10 */
  long seed;
  double *x, sttime, endtime;
  void (*fun)(int, double *);        /* Pointer to sorting function */
  seed = time(NULL); /* Get current time, in case seed is not given */
  if (argc > 1)                /* First character of first argument */
    method = *argv[1];       /* denotes the employed sorting method */
  if (argc > 2)
    n = atoi(argv[2]);     /* Second argument is number of elements */
  if (argc > 3)                         /* Third argument is seed for */
    seed = atoi(argv[3]);                  /* random number generator */
  switch(method) {        /* Prepare calling the appropriate method */
    case 'b':
      fun = bubblesort; name = "bubblesort"; break;
    case 's':
      fun = selectsort; name = "selectsort"; break;
    case 'i':
      fun = insertsort; name = "insertsort"; break;
    case 'q':
      fun = quicksort; name = "quicksort"; break;
    default:
      printf("Sorry, no such method\n");
      return 1;
  }                                  /* Allocate memory for the array */
  if ((x = malloc(n * sizeof(double))) == NULL) {
    printf("Sorry, not enough memory\n");
    return 1; }
```

```
  srand((unsigned int) seed); /* Initialize random number generator */
  for (i=0 ; i < n ; i++) /* Generate double floating point numbers */
    x[i] = ((double) rand())/RAND_MAX;
  printf("Random numbers\n");
  for (i=0 ; i < n ; i++) {
    printf("%6.4f ", x[i]);                        /* Print them out */
    if (i%10 == 9)                                   /* 10 in a line */
      printf("\n"); }
  printf("\n");
  printf("Sorting by %s\n", name);
  sttime = ((double) clock())/CLOCKS_PER_SEC;       /* Get CPU time */
                                /* consumed since start of program */
  (*fun)(n, x);                                  /* Call sorting method */
  endtime = ((double) clock())/CLOCKS_PER_SEC;    /* Again CPU time */
                             /* Difference endtime-sttime should be */
                                   /* CPU time consumed for sorting */
  for (i=0 ; i < n ; i++) {
    printf("%6.4f ", x[i]);                /* Print out sorted array */
    if (i%10 == 9)
      printf("\n"); }
  printf("\n");             /* Print out CPU time needed for sorting */
  printf("Time: %.2f secs\n", endtime-sttime);
  free(x); return 0;
}

void bubblesort(int n, double *x)
{ int i, j;
  for (i=1 ; i <= n-1 ; i++)          /* Bring appropriate element, */
                                 /* that is the bubble, to place i-1 */
    for (j=n-1 ; j >= i ; j--)
      if (x[j-1] > x[j])     /* Compare pairwise from bottom to top */
        swapd(&x[j-1], &x[j]);                /* and swap if needed */
}

void selectsort(int n, double *x)
{ int i, j, min;
  for (i=1 ; i <= n-1 ; i++) {
    min = i-1;             /* Let current minimum be the i-1 element */
    for (j=i ; j <= n-1 ; j++)
      if (x[j] < x[min])  /* Check if any element after i-1 is less */
        min = j; /* than so far minimum and make it the new minimum */
    swapd(&x[i-1], &x[min]); } /* Exchange minimum with i-1 element */
}
```

```
void insertsort(int n, double *x)
{ int i, j;
  for (i=1 ; i <= n-1 ; i++) {      /* Insert element at place i in */
                       /* its correct position from places 0 to i-1 */
    j = i-1;
    while (j >= 0 && x[j] > x[j+1]) {/* Move repeatedly the element */
      swapd(&x[j], &x[j+1]);                    /* until it reaches */
      j--; } }                               /* its correct position */
}

void quicksort(int n, double *x)
{ quicksort_body(x, 0, n-1);    /* Call recursive quicksort to sort */
}          /* elements of the array from position 0 to position n-1 */

void quicksort_body(double *x, int up, int down)
{ int start, end;
  start = up;              /* Save start position of small elements */
  end = down;                /* Save end position of large elements */
  while (up < down) {            /* Pivot element is at up position */
    while (x[down] >= x[up] && up < down)      /* Let down elements */
      down--;              /* larger than pivot stay where they are */
    if (up != down) {                    /* If pivot is not reached */
      swapd(&x[up], &x[down]);   /* echange it with smaller element */
      up++;     /* Pivot is at down position, move up a bit further */
    }
    while (x[up] <= x[down] && up < down)        /* Let up elements */
    up++;                 /* smaller than pivot stay where they are */
    if (up != down) {                    /* If pivot is not reached */
      swapd(&x[up], &x[down]);   /* exchange it with larger element */
      down--;   /* Pivot is at up position, move down a bit further */
    } }       /* Now up = down is the position of the pivot element */
  if (start < up-1) /* Is there at least one element left of pivot? */
    quicksort_body(x, start, up-1); /* Quick(sort) smaller elements */
  if (end > down+1)/* Is there at least one element right of pivot? */
    quicksort_body(x, down+1, end);  /* Quick(sort) larger elements */
}

void swapd(double *a, double *b)       /* Just exchange two doubles */
{ double temp;
  temp = *a;
  *a = *b;
  *b = temp; }
```

```
% gcc -o sorting sorting.c
% ./sorting b 30
Random numbers
0.1914 0.8545 0.6266 0.4347 0.0597 0.1278 0.1717 0.9190 0.8703 0.5835
0.5544 0.2181 0.4099 0.3580 0.7294 0.7789 0.1550 0.3378 0.4779 0.8070
0.2298 0.0863 0.6197 0.7538 0.2502 0.4336 0.5880 0.5354 0.6012 0.6745

Sorting by bubblesort
0.0597 0.0863 0.1278 0.1550 0.1717 0.1914 0.2181 0.2298 0.2502 0.3378
0.3580 0.4099 0.4336 0.4347 0.4779 0.5354 0.5544 0.5835 0.5880 0.6012
0.6197 0.6266 0.6745 0.7294 0.7538 0.7789 0.8070 0.8545 0.8703 0.9190

Time: 0.00 secs
% ./sorting s 30
Random numbers
0.1520 0.1810 0.6908 0.9386 0.9103 0.1460 0.6984 0.2948 0.5047 0.7014
0.2916 0.7622 0.9437 0.1535 0.0158 0.2708 0.6973 0.9403 0.3101 0.8563
0.0342 0.6943 0.6948 0.5267 0.4925 0.9227 0.8265 0.4250 0.9225 0.0146

Sorting by selectsort
0.0146 0.0158 0.0342 0.1460 0.1520 0.1535 0.1810 0.2708 0.2916 0.2948
0.3101 0.4250 0.4925 0.5047 0.5267 0.6908 0.6943 0.6948 0.6973 0.6984
0.7014 0.7622 0.8265 0.8563 0.9103 0.9225 0.9227 0.9386 0.9403 0.9437

Time: 0.00 secs
% ./sorting i 30
Random numbers
0.1593 0.5741 0.8110 0.1777 0.4501 0.6941 0.7238 0.8614 0.1461 0.7922
0.8593 0.8994 0.7509 0.0171 0.4619 0.9863 0.9694 0.7962 0.1611 0.9896
0.5480 0.8230 0.6144 0.6332 0.5996 0.2770 0.3051 0.2945 0.5684 0.7232

Sorting by insertsort
0.0171 0.1461 0.1593 0.1611 0.1777 0.2770 0.2945 0.3051 0.4501 0.4619
0.5480 0.5684 0.5741 0.5996 0.6144 0.6332 0.6941 0.7232 0.7238 0.7509
0.7922 0.7962 0.8110 0.8230 0.8593 0.8614 0.8994 0.9694 0.9863 0.9896

Time: 0.00 secs
% ./sorting q 30
Random numbers
0.1687 0.4653 0.4296 0.9194 0.4879 0.2398 0.2521 0.4263 0.7844 0.3863
0.9253 0.5324 0.5617 0.8795 0.9027 0.7057 0.7409 0.1454 0.0162 0.1232
0.5537 0.9557 0.0353 0.7299 0.2103 0.1340 0.7720 0.6671 0.2188 0.4183

Sorting by quicksort
0.0162 0.0353 0.1232 0.1340 0.1454 0.1687 0.2103 0.2188 0.2398 0.2521
0.3863 0.4183 0.4263 0.4296 0.4653 0.4879 0.5324 0.5537 0.5617 0.6671
0.7057 0.7299 0.7409 0.7720 0.7844 0.8795 0.9027 0.9194 0.9253 0.9557

Time: 0.00 secs
%
```

```
% hostname
linux29
% ./sorting b 20000 2016 | tail -4
0.9987 0.9987 0.9987 0.9987 0.9987 0.9988 0.9988 0.9989 0.9991 0.9991
0.9992 0.9992 0.9992 0.9992 0.9993 0.9993 0.9993 0.9995 0.9996 1.0000

Time: 2.07 secs
% ./sorting b 50000 2016 | tail -4
0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9995 0.9995 0.9995
0.9995 0.9995 0.9996 0.9996 0.9997 0.9997 0.9998 0.9999 0.9999 1.0000

Time: 12.87 secs
% ./sorting s 20000 2016 | tail -4
0.9987 0.9987 0.9987 0.9987 0.9987 0.9988 0.9988 0.9989 0.9991 0.9991
0.9992 0.9992 0.9992 0.9992 0.9993 0.9993 0.9993 0.9995 0.9996 1.0000

Time: 0.78 secs
% ./sorting s 50000 2016 | tail -4
0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9995 0.9995 0.9995
0.9995 0.9995 0.9996 0.9996 0.9997 0.9997 0.9998 0.9999 0.9999 1.0000

Time: 4.85 secs
% ./sorting i 20000 2016 | tail -4
0.9987 0.9987 0.9987 0.9987 0.9987 0.9988 0.9988 0.9989 0.9991 0.9991
0.9992 0.9992 0.9992 0.9992 0.9993 0.9993 0.9993 0.9995 0.9996 1.0000

Time: 1.02 secs
% ./sorting i 50000 2016 | tail -4
0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9995 0.9995 0.9995
0.9995 0.9995 0.9996 0.9996 0.9997 0.9997 0.9998 0.9999 0.9999 1.0000

Time: 6.36 secs
% ./sorting q 20000 2016 | tail -4
0.9987 0.9987 0.9987 0.9987 0.9987 0.9988 0.9988 0.9989 0.9991 0.9991
0.9992 0.9992 0.9992 0.9992 0.9993 0.9993 0.9993 0.9995 0.9996 1.0000

Time: 0.01 secs
% ./sorting q 50000 2016 | tail -4
0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9994 0.9995 0.9995 0.9995
0.9995 0.9995 0.9996 0.9996 0.9997 0.9997 0.9998 0.9999 0.9999 1.0000

Time: 0.01 secs
% ./sorting q 3000000 2016 | tail -4
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000

Time: 0.84 secs
%
```

Αναζήτηση σε πίνακες

- Πολλές φορές έχουμε ένα πίνακα από στοιχεία (αριθμούς, συμβολοσειρές, οτιδήποτε) και θέλουμε να δούμε αν ένα συγκεκριμένο στοιχείο, ίδιου τύπου με αυτά του πίνακα, ανήκει στον πίνακα. Η διαδικασία αυτή λέγεται αναζήτηση.
- Ο απλούστερος τρόπος να κάνουμε αναζήτηση σ΄ ένα πίνακα είναι με τη λεγόμενη σειριακή αναζήτηση. Διατρέχουμε τα στοιχεία του πίνακα, ένα προς ένα, μέχρι να βρούμε το στοιχείο που ψάχνουμε, αν το βρούμε τελικά.
- Αν ο πίνακας είναι μεγάλος, η σειριακή αναζήτηση μπορεί να είναι πολύ χρονοβόρα. Αν όμως ταξινομήσουμε τον πίνακα, έστω σε αύξουσα σειρά, μπορούμε να εφαρμόσουμε μία πολύ ταχύτερη μέθοδο, τη δυαδική αναζήτηση.
- Με τη δυαδική αναζήτηση, συγκρίνουμε το στοιχείο που ψάχνουμε με το μεσαίο (ή ένα από τα δύο μεσαία) στοιχείο του πίνακα. Αν συμπίπτει, σταματάμε την αναζήτηση αφού το στοιχείο βρέθηκε. Αν όχι, στην περίπτωση που το στοιχείο προηγείται του μεσαίου, ψάχνουμε να το βρούμε στο πρώτο μισό του πίνακα. Αν έπεται του μεσαίου, το ψάχνουμε στο δεύτερο μισό. Η αναζήτηση στον κατάλληλο υποπίνακα γίνεται με τον ίδιο τρόπο, συγκρίνοντας το στοιχείο με το μεσαίο του υποπίνακα και η διαδικασία συνεχίζει με αυτόν τον τρόπο, μέχρις ότου είτε να βρεθεί το στοιχείο είτε να φτάσουμε σε κενό υποπίνακα.
- Ποιες είναι οι πολυπλοκότητες των μεθόδων; [α΄]

[α΄] $O(n)$ και $O(\log n)$, για τη σειριακή και τη δυαδική αναζήτηση, αντίστοιχα.

Μέθοδοι αναζήτησης

```
/* File: searching.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void heapsort(int, char **);
void heapify(char **, int, int);
int seqsearch(char *, int, char **);
int binsearch(char *, int, char **);
void swapwords(char **, char **);

int main(int argc, char *argv[])
{ int k = 0;                    /* Counter of words that will be read */
  int nmax = 1000;       /* Default maximum number of words to read */
  double sttime, endtime;
  char search = 's';                /* Default is sequential search */
  char **words, *arg, buf[81];
  while (--argc) {
    arg = *++argv;
    if (!strcmp(arg, "-max")) {               /* Get maximum number */
      if (argc > 1 && --argc)                   /* of words to read */
        nmax = atoi(*++argv); }
    else if (!strcmp(arg, "-seq"))      /* Select sequential search */
      search = 's';
    else if (!strcmp(arg, "-bin"))          /* Select binary search */
      search = 'b';
    else if (!strcmp(arg, "-words")) {  /* Give words to search for */
      argc--;
      break; } }
  argv++;    /* Allocate memory to store addresses of words to read */
  if ((words = malloc(nmax * sizeof(char *))) == NULL) {
    fprintf(stderr, "Not enough memory\n");
    return 1; }
```

```
  while (k < nmax && scanf("%80s", buf) != EOF) {
              /* Read words until EOF or maximum number reached and */
                                  /* allocate memory to store them */
    if ((words[k] = malloc((strlen(buf)+1) * sizeof(char))) == NULL) {
      fprintf(stderr, "Not enough memory\n");
      return 2; }
    strcpy(words[k++], buf); }                    /* Store word read */
  if (search == 'b')                    /* If binary search selected */
    heapsort(k, words);       /* sort words via the heapsort method */
  sttime = ((double) clock())/CLOCKS_PER_SEC;  /* Search start time */
  while (argc--) {
    arg = *argv++;
    switch (search) {
      case 's':                         /* The sequential search case */
        printf("%sfound %s\n",
          seqsearch(arg, k, words) ? "    " : "not ", arg);
        break;
      case 'b':                              /* The binary search case */
        printf("%sfound %s\n",
          binsearch(arg, k, words) ? "    " : "not ", arg);
        break; } }
  endtime = ((double) clock())/CLOCKS_PER_SEC;   /* Search end time */
  printf("Searching time is %.2f seconds\n", endtime-sttime);
  return 0; }

void heapsort(int n, char **x)
{ int i;
  for (i=(n/2)-1 ; i >= 0 ; i--)       /* Transform array to a heap */
/* A heap is an implicit binary tree where each node is not smaller */
    heapify(x, i, n-1);            /* than its immediate successors */
  for (i=n-1 ; i >= 1 ; i--) {/* Move heap root to bottom rightmost */
    swapwords(&x[0], &x[i]);    /* position not already settled and */
    heapify(x, 0, i-1); }/*transform to a heap the rest of the tree */
}
```

```
void heapify(char **x, int root, int bottom)
{ int maxchild; /* Transform to a heap the subtree starting at root */
                                          /* up to element bottom */
  while (2*root < bottom) {         /* Do we have still work to do? */
    if (2*root+1 == bottom)    /* If left child is last to consider */
      maxchild = 2*root+1;             /* this is the maximum child */
    else if (strcmp(x[2*root+1], x[2*root+2]) > 0)     /* Otherwise */
      maxchild = 2*root+1;           /* select maximum between left */
    else
      maxchild = 2*root+2;               /* and right child of root */
    if (strcmp(x[maxchild], x[root]) > 0) {/* Compare maximum child */
      swapwords(&x[maxchild], &x[root]);      /* with root and swap */
      root = maxchild; } /* accordingly, defining also the new root */
    else
      break; } }                              /* OK, we made our heap */

int seqsearch(char *w, int n, char **x)
{ int i;
  for (i=0 ; i < n ; i++)
    if (!strcmp(w, x[i]))       /* So simple, what to comment here! */
      return 1;
  return 0; }

int binsearch(char *w, int n, char **x)
{ int cond, low, high, mid;
  low = 0;                                /* Lower limit for searching */
  high = n-1;                             /* Upper limit for searching */
  while (low <= high) {          /* Do we have space for searching? */
    mid = (low+high)/2;         /* Medium element of search interval */
                          /* to compare with word we are looking for */
    if ((cond = strcmp(w, x[mid])) < 0)   /* Compare medium to word */
      high = mid-1;       /* Not found, word might be at first half */
    else if (cond > 0)
      low = mid+1;       /* Not found, word might be at second half */
    else return 1; }                                  /* We found it! */
  return 0; }                               /* Sorry, word not found */

void swapwords(char **w1, char **w2)
{ char *temp;  /* The well-known swap function for the strings case */
  temp = *w1;
  *w1 = *w2;
  *w2 = temp; }
```

```
% gcc -o searching searching.c
% cat test_words.txt
Hello there! How are you?
Hello!!! I am fine. What about you? Are you OK?
Yes, I am fine. Thank you.
% ./searching -seq -max 50 -words you word fine I < test_words.txt
    found you
not found word
not found fine
    found I
Searching time is 0.00 seconds
% ./searching -bin -max 50 -words you word fine I < test_words.txt
    found you
not found word
not found fine
    found I
Searching time is 0.00 seconds
% ./searching -bin -max 12 -words you word fine I < test_words.txt
not found you
not found word
not found fine
    found I
Searching time is 0.00 seconds
% wc -w /usr/share/dict/words
234937 /usr/share/dict/words
% ./searching -seq -max 240000 \
?       -words `cat KRExcerpt.txt` < /usr/share/dict/words | tail
    found well
    found to
    found write
    found major
not found programs
    found in
    found many
    found different
not found domains.
Searching time is 0.08 seconds
% ./searching -bin -max 240000 \
?       -words `cat KRExcerpt.txt` < /usr/share/dict/words | tail
    found well
    found to
    found write
    found major
not found programs
    found in
    found many
    found different
not found domains.
Searching time is 0.00 seconds
%
```

```
% hostname
linux29
% wc -w big.txt
1095695 big.txt
% ./searching -seq -max 1100000 \
?       -words `cat KRExcerpt.txt` < big.txt | tail -6
    found programs
    found in
    found many
    found different
not found domains.
Searching time is 0.14 seconds
% ./searching -bin -max 1100000 \
?       -words `cat KRExcerpt.txt` < big.txt | tail -6
    found programs
    found in
    found many
    found different
not found domains.
Searching time is 0.00 seconds
% ./searching -bin -max 1100000 \
?       -words `head -180000 /usr/share/dict/words` < big.txt \
?                                           | grep 'not found' | wc -l
167331
% ./searching -bin -max 1100000 \
?       -words `head -180000 /usr/share/dict/words` < big.txt \
?                                           | grep '    found' | wc -l
12669
% ./searching -bin -max 1100000 \
?       -words `head -20000 /usr/share/dict/words` < big.txt \
?                                           | grep 'Searching time'
Searching time is 0.01 seconds
% ./searching -seq -max 1100000 \
?       -words `head -20000 /usr/share/dict/words` < big.txt \
?                                           | grep 'Searching time'
Searching time is 209.13 seconds
%
```

Ένα πρόγραμμα C πρέπει να είναι ...

- ... **σωστό**. Προφανώς! Πρόγραμμα που δίνει λάθος αποτελέσματα πρέπει να διορθωθεί.
- ... **αποδοτικό**. Τα προγράμματα πρέπει να δίνουν τα αποτελέσματά τους μέσα σε εύλογο χρονικό διάστημα. Βέβαια, ο ορισμός του "εύλογου" ποικίλει ανάλογα με την περίπτωση και τις συνθήκες. Σε κάθε περίπτωση, θα πρέπει να προσπαθούμε το πρόγραμμά μας να έχει τη μέγιστη δυνατή απόδοση. Επίσης, θα πρέπει να ελέγχουμε τη συμπεριφορά του και για, πιθανώς ασυνήθιστα, μεγάλες εισόδους. Πάντως, ένα πρόγραμμα που θεωρητικά κάποια στιγμή θα τερματίσει, αλλά μπορεί να μην ζούμε για να το δούμε, δεν έχει πρακτική αξία.
- ... **εύρωστο**. Δεν πρέπει το πρόγραμμα να βγάζει λάθη εκτέλεσης, οποιαδήποτε και αν είναι η είσοδος που δέχτηκε. Δεν αρκεί να συμπεριφέρεται ομαλά μόνο για τις αναμενόμενες εισόδους.
- ... **τεκμηριωμένο**. Η τεκμηρίωση συνίσταται στην καλή επιλογή ονομάτων μεταβλητών, συναρτήσεων, κλπ., καθώς και στην προσθήκη σχολίων. Η τεκμηρίωση πρέπει να είναι τόση όση χρειάζεται για να κάνει το πρόγραμμα κατανοητό. Ούτε λιγότερη, αλλά ούτε και περισσότερη.
- ... **ευανάγνωστο**. Πρέπει να έχει συνεπή στοίχιση, να αποφεύγονται γραμμές με περισσότερους από 80 χαρακτήρες, να υπάρχουν κενά γύρω από τελεστές, όπου αυτό βελτιώνει την αναγνωσιμότητα.

Συχνά προγραμματιστικά λάθη στην C

- Να κάνουμε διαίρεση μεταξύ ακεραίων, ευελπιστώντας ότι θα πάρουμε σαν αποτέλεσμα έναν αριθμό κινητής υποδιαστολής, ενώ στην πραγματικότητα παίρνουμε το ακέραιο πηλίκο της διαίρεσης.
- Να νομίσουμε ότι σε μία ακέραια μεταβλητή μπορούμε να φυλάξουμε τιμή οσοδήποτε μεγάλη, ή ότι σε μία μεταβλητή κινητής υποδιαστολής μπορούμε να έχουμε όση ακρίβεια θέλουμε.
- Να κάνουμε λάθος στην οριακή συνθήκη τερματισμού μίας δομής επανάληψης.
- Να χρησιμοποιήσουμε για τελεστή σύγκρισης μέσα σε συνθήκη το =, αντί για το ==.
- Να βάλουμε μετά από μία `for` ή μία `while` ένα `;` πριν την εντολή ή το μπλοκ εντολών που αποτελούν το σώμα τους.
- Να ξεχάσουμε ότι τα στοιχεία ενός πίνακα με διάσταση `N` αριθμούνται από το `0` έως το `N-1`, και όχι από το `1` έως το `N`.
- Να χειριστούμε απρόσεκτα τους δείκτες, για παράδειγμα να δοκιμάσουμε να γράψουμε εκεί που πιστεύουμε ότι δείχνει ένας δείκτης, χωρίς να έχουμε κάνει την απαιτούμενη δέσμευση μνήμης.
- Να δεσμεύουμε δυναμικά μνήμη με επαναληπτικό τρόπο και να μην την αποδεσμεύουμε όταν δεν την χρειαζόμαστε.

- Να βάλουμε `;` στο τέλος μίας οδηγίας `#define`.
- Να ξεχάσουμε να βάλουμε παρενθέσεις γύρω από το κείμενο αντικατάστασης μίας μακροεντολής, που ορίζουμε με `#define`, και όπου αλλού χρειάζεται μέσα σ΄ αυτό.
- Να παρεξηγήσουμε τις προτεραιότητες των τελεστών, για παράδειγμα να γράψουμε

  ```
  while (ch = getchar() != EOF)
  ```

  αντί για

  ```
  while ((ch = getchar()) != EOF)
  ```

- Να κάνουμε σύγχυση ανάμεσα στην έννοια της προτεραιότητας των τελεστών, που έχει να κάνει με το πού εφαρμόζεται ένας τελεστής, με τη σειρά εκτέλεσης των ενεργειών που προσδιορίζουν οι τελεστές. Για παράδειγμα, το `*p++` είναι όντως το ίδιο με το `*(p++)`, το οποίο όμως πολλοί νομίζουν, λάθος, ότι σημαίνει ότι πρώτα θα αυξηθεί ο δείκτης `p` και μετά θα προσπελάσουμε το περιεχόμενο της νέας διεύθυνσης. **ΟΧΙ**. Απλώς, ο τελεστής `++` εδώ είναι μεταθεματικός, δηλώνοντας ότι πρώτα θα προσπελασθεί το περιεχόμενο της διεύθυνσης που δείχνει ο `p` και μετά θα αυξηθεί αυτός. Οι παρενθέσεις γύρω από το `p++` δείχνουν πού εφαρμόζεται ο τελεστής `++` (στον δείκτη `p`) και όχι πότε θα εκτελεσθεί η πράξη που υποδεικνύει.

- Να χειριστούμε απρόσεκτα μεταβλητές με το ίδιο όνομα, παρεξηγώντας τις εμβέλειες καθεμιάς, και να χρησιμοποιούμε κάποια απ΄ αυτές, ενώ πραγματικά χρησιμοποιούμε κάποια άλλη.
- Να θεωρήσουμε ότι τοπικές μεταβλητές είναι αρχικοποιημένες (για παράδειγμα, σε 0), παρότι εμείς δεν έχουμε κάνει ρητά κάτι τέτοιο.
- Να διατυπώσουμε απρόσεκτα εμφωλευμένες εντολές `if`, παραλείποντας άγκιστρα { και } σε περιπτώσεις που χρειάζονται, με αποτέλεσμα η εντολή να είναι μεν συντακτικά σωστή, οπότε ο μεταγλωττιστής δεν διαμαρτύρεται, αλλά να σημαίνει κάτι άλλο από αυτό που είχαμε πρόθεση να γράψουμε.
- Να ξεχάσουμε κρίσιμα `break` στις περιπτώσεις μίας εντολής `switch`.
- Να ταυτίσουμε το `'A'` με το `"A"`.
- Να παραβλέψουμε το γεγονός ότι μία συμβολοσειρά τερματίζει με τον χαρακτήρα `'\0'`.
- Να συγκρίνουμε συμβολοσειρές με τον τελεστή ==, αντί με τη συνάρτηση `strcmp`.
- Να μην βάλουμε συνθήκη τερματισμού μίας αναδρομής.
- Να καλέσουμε την `scanf` περνώντας σ΄ αυτήν τις μεταβλητές που θέλουμε να διαβάσουμε, αντί για τις διευθύνσεις τους.

Βιβλιογραφία

- Brian W. Kernighan, Dennis M. Ritchie: "*Η Γλώσσα Προγραμματισμού C*", Prentice Hall (Ελληνική μετάφραση, εκδόσεις Κλειδάριθμος), 1988.
- Γ. Σ. Τσελίκης, Ν. Δ. Τσελίκας: "*C: Από τη Θεωρία στην Εφαρμογή*", 3η έκδοση, 2016.
- Νικόλαος Μισυρλής: "*Εισαγωγή στον Προγραμματισμό με την C*", 3η έκδοση, 2007.
- Νίκος Μ. Χατζηγιαννάκης: "*Η Γλώσσα C σε Βάθος*", 5η έκδοση, Εκδόσεις Κλειδάριθμος, 2017.
- Jeri R. Hanly, Elliot B. Koffman: "*Αρχές και Τεχνικές Προγραμματισμού με τη Γλώσσα C*", Pearson Education (Ελληνική μετάφραση, εκδόσεις Κριτική), 1η έκδοση, 2021.
- Δημήτριος Καρολίδης: "*Μαθαίνετε Εύκολα C*", 2η έκδοση, 2021.
- Herbert Schildt: "*Οδηγός της C*", 3η έκδοση, McGraw-Hill (Ελληνική μετάφραση, εκδόσεις Γκιούρδας), 2004.
- Eric S. Roberts: "*Η Τέχνη και Επιστήμη της C*", Addison-Wesley (Ελληνική μετάφραση, εκδόσεις Κλειδάριθμος), 2004.
- Brian W. Kernighan, Rob Pike: "*The Practice of Programming*", Addison-Wesley, 1999.
- Jon Bentley: "*Programming Pearls*", 2nd edition, Addison-Wesley, 2000.
- Steven S. Skiena: "*The Algorithm Design Manual*", Springer-Verlag, 1998.
- H. M. Deitel, P. J. Deitel: "*C: How to Program*", 3rd edition, Prentice Hall, 2001.
- Νικόλαος Θ. Αποστολάτος: "*Προγραμματισμός – Εφαρμογές*", 1995.
- Διομήδης Σπινέλλης: "*Ανάγνωση Κώδικα – Η Προοπτική του Ανοικτού Λογισμικού*", εκδόσεις Κλειδάριθμος, 2005.
- Brian W. Kernighan, Rob Pike: "*Το Περιβάλλον Προγραμματισμού Unix*", Prentice Hall (Ελληνική μετάφραση, εκδόσεις Κλειδάριθμος), 1989.
- Donald E. Knuth: "*The Art of Computer Programming: Sorting and Searching (Vol. 3)*", 2nd edition, Addison-Wesley, 1998.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: "*Introduction to Algorithms*", 2nd edition, The MIT Press, 2001.

Ευχαριστίες ...

- ... στην Επίκουρη Καθηγήτρια *Ιζαμπώ Καράλη* για τη μετάδοση πολύτιμης εμπειρίας, ιδεών και υλικού από το μάθημα του Αντικειμενοστραφούς Προγραμματισμού που διδάσκει στο Τμήμα μας από το ακαδημαϊκό έτος 2002–03.
- ... επίσης στην *Ιζαμπώ* και στους απόφοιτους του Τμήματος (προπτυχιακού και μεταπτυχιακού κύκλου) *Στέφανο Σταμάτη* και *Νίκο Ποθητό* που διάβαζαν με απεριόριστη υπομονή τα πρωτόλεια αυτών των διαφανειών/σημειώσεων, ενόσω γραφόντουσαν, και υπέδειξαν ένα πλήθος από λάθη και παραλείψεις που είχαν, κάνοντας ταυτόχρονα και προτάσεις για τη διόρθωσή τους.

Παναγιώτης Σταματόπουλος