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.

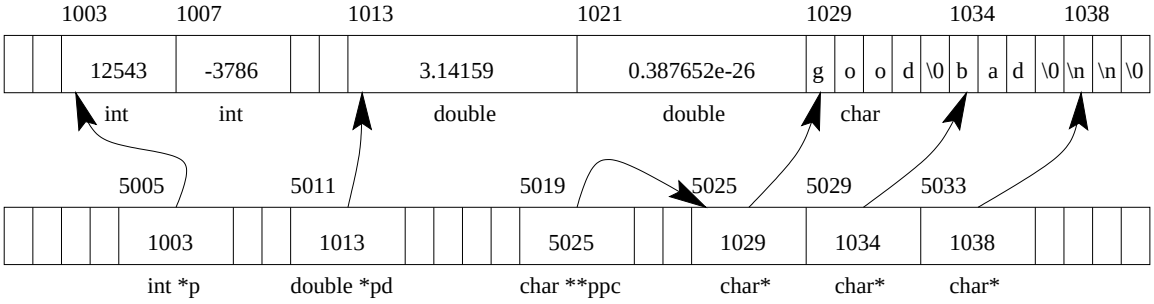
- Περί δεικτών, αλλά και πινάκων, που θα συζητήσουμε στη συνέχεια, το Κεφάλαιο 5 από το [KR] (σελ. 135–178) είναι μία ανεξάντλητη πηγή πληροφοριών. [α΄]

[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.

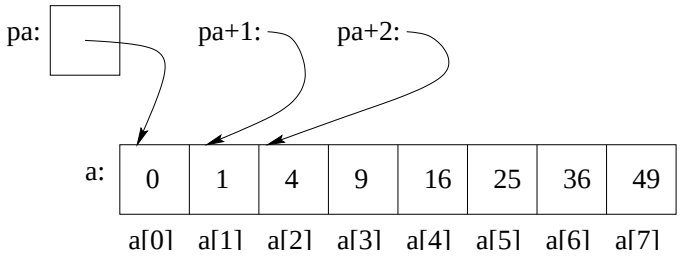
- Για παράδειγμα, αν έχουμε ορίσει έναν πίνακα `a`, τότε η έκφραση `*(a+i)` είναι ισοδύναμη με την `a[i]`. Ομοίως, και οι εκφράσεις `a+i` και `&a[i]` είναι ισοδύναμες.

- 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/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.

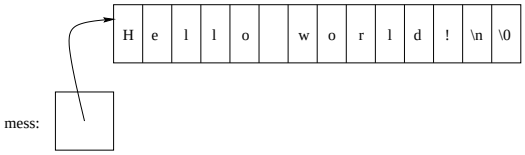
Συμβολοσειρές

- 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 πίνακας

- 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 διδιάστατο 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 διδιάστατο 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 διδιάστατος 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 διδιάστατο 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].