ΚΕΦΑΛΑΙΟ 1

Ο ΠΡΟΕΠΕΞΕΡΓΑΣΤΗΣ ΤΗΣ C

Ο προεπεξεργαστής «παραλαμβάνει» το πρόγραμμα πριν τον μεταγλωττιστή και μεταβάλει το πηγαίο πρόγραμμα σύμφωνα με τις λεγόμενες «*οδηγίες*» προς αυτόν. Οι οδηγίες προς τον προεπεξεργαστή αρχίζουν με το σύμβολο # και μπορούν να χρησιμοποιηθούν οπουδήποτε μέσα στο πρόγραμμα.

1.1. Η ΟΔΗΓΙΑ #define

Χρησιμοποιείται:

1.1.1. Για τη δημιουργία σταθερών. Στο παρακάτω παράδειγμα, η σταθερά PI ορίζεται με τιμή 3.14159:

```
#define PI 3.14159
.....
int  r;
area = 2 * PI * r ;
```

Κατά τη διάρκεια της προεπεξεργασίας, το όνομα που ορίζεται με την #define αντικαθίσταται με την τιμή του, άρα η εντολή που υπολογίζει το area στον παραπάνω πηγαίο κώδικα γίνεται:

```
area = 2 * 3.14259 * r;
```

Με την #define δεν δημιουργούμε μόνο σταθερές. Στο παρακάτω παράδειγμα, η εντολή printf μέσα στο πρόγραμμα αντικαθίσταται με την εντολή DISPLAY_ON_SCREEN:

```
#defineDISPLAY_ON_SCREENprintf
```

Μετά από την εντολή αυτή, δείτε πώς μπορούμε να εμφανίσουμε την τιμή του ακεραίου ak:

```
DISPLAY_ON_SCREEN("%d", r);
```

Μια σταθερά μπορεί να υπάρχει στον ορισμό μιας άλλης σταθεράς. Έτσι για παράδειγμα, το AK μετά τα παρακάτω έχει τιμή 50:

```
#define TR  50
#define AK  TR
```

1.1.2. Για τη δημιουργία μακροεντολών. Στο παρακάτω παράδειγμα, στην οθόνη γράφεται ο μέσος όρος των a, b και c.

```
#define   MESOS_OROS(x, y, z)  (x+y+z)/3
.....
int a, b, c, mo;
```

```
mo = MESOS_OROS (a, b, c);
printf ("%d", mo);
```

Μια μακροεντολή μπορεί να έχει οσεσδήποτε παραμέτρους, πρέπει να έχουμε όμως ὑπ᾽ όψη τους εξής περιορισμούς:

- Όλες οι παράμετροι της μακροεντολής πρέπει να υπάρχουν και στην παράσταση με την οποία θα αντικατασταθεί η μακροεντολή.
- Η αριστερή παρένθεση της μακροεντολής πρέπει να ακολουθεί αμέσως μετά το όνομά της, χωρίς κενό ανάμεσα.

Μια μακροεντολή είναι φανερό ότι μπορεί να χρησιμοποιηθεί όπως μια συνάρτηση, κυρίως όταν ο κώδικάς της είναι σύντομος. Είναι ταχύτερη σε εκτέλεση από μια συνάρτηση, αλλά υπάρχει «επιβάρυνση» στο μέγεθος του κώδικα του προγράμματος, αφού ο κώδικάς της ενσωματώνεται στο πρόγραμμα σε όλα τα σημεία που καλείται.

Μια μακροεντολή μπορεί να επεκτείνεται σε περισσότερες από μια γραμμές, αρκεί στο τέλος της κάθε γραμμής να υπάρχει ο χαρακτήρας \. Στο παρακάτω, το ΑΚ παίρνει τιμή 55

```
#define  AK 10\
           +20\
           +25
```

Το # μπροστά από το όρισμα μιας μακροεντολής δημιουργεί μια συμβολοσειρά με όνομα αυτό του ορίσματος. Το επόμενο παράδειγμα θα εμφανίσει στην οθόνη:

```
str = text
```

(Το παράδειγμα είναι από το βιβλίο «C Από τη θεωρία στην Εφαρμογή», Γ.Σ. Τσελίκης, Ν.Δ. Τσελίκας, ISBN: 978-960-93-1961-4).

```
#define   f(s) printf ("%s = %s\n" #s, s);
.....
char *str = "text";
f (str);
```


1.2. ΜΕΤΑΓΛΩΤΤΙΣΗ ΥΠΟ ΣΥΝΘΗΚΗ.

1.2.1. Οιοδηγίες #if, #elif, #else, #endif.

- Η οδηγία **#if** λειτουργεί όπως η εντολή if, με τη διαφορά ότι η #if ελέγχει εάν μεταγλωτίζονται κάποιες εντολές. Την #if ακολουθεί ένα μπλόκ εντολών, χωρίς άγκιστρα. Συνήθως χρησιμοποιείται για έλεγχο ήδη ορισμένων σταθερών. Δεν επιτρέπεται ο έλεγχος float τιμών.

- Η εντολή **#else** είναι η αντίστοιχη της else. Η ύπαρξή της δεν είναι υποχρεωτική.
- Η **#elif** είναι αντίστοιχη της else...if. Η ύπαρξή της δεν είναι υποχρεωτική.
- Η **#endif** σηματοδοτεί το τέλος της #if και είναι υποχρεωτική.

Στον παρακάτω πηγαίο κώδικα η μακροεντολή power υπολογίζει την τιμή του x^2 ή του x^3, ανάλογα με την τιμή της σταθεράς NUM:

```
#define NUM 3
#if NUM==2
#define power (x) (x*x)
#elif NUM==3
#define power (x) (x*x*x)
#endif
```

1.2.2. Η οδηγία #ifdef. Η χρήση της δίνει αποτέλεσμα αληθές ή ψευδές, ανάλογα με το εάν έχει οριστεί ήδη μια σταθερά ή όχι. Η οδηγία θα δώσει τιμή αληθή ακόμη και εάν δεν έχει οριστεί συγκεκριμένη τιμή στην σταθερά. Με λίγα λόγια, μετά τα παρακάτω, η #ifdef δίνει τιμή αληθή:

```
#define NUM
#ifdef NUM
```

Το παράδειγμα που ακολουθεί αποτελεί μια τροποποίηση του παραπάνω παραδείγματος της παραγράφου 1.2.1. Εδώ, εάν η τιμή του NUM δεν έχει οριστεί, εάν λείπει δηλαδή η οδηγία #define NUM 3, η μακροεντολή power έχει ως αποτέλεσμα την τιμή του x:

```
#define NUM 3
#ifdef NUM
#if NUM==2
#define power (x) (x*x)
#elif NUM==3
#define power (x) (x*x*x)
#endif
#else
#define power(x) x
#endif
```

Η εντολή #ifdef χρειάζεται επίσης σηματοδότηση του τέλους της με την εντολή #endif (όχι προφανώς η #elif).

1.2.3. Η οδηγία #ifndef. Είναι η «αντίστροφη» οδηγία της #ifdef. Η χρήση της οδηγίας αυτής δίνει αποτέλεσμα αληθές εάν δεν έχει οριστεί ήδη μια σταθερά. Απαιτείται και εδώ η ύπαρξη της #endif. Δείτε το παράδειγμα που ακολουθεί:

```
#define NUM 5
#ifndef NUM
#define apot(x) x
#else
#define apot(x) (1.0*x/NUM)
#endif
```

1.2.4. Η οδηγία #error.Τερματίζει την μεταγλώττιση του προγράμματος, μπορεί δε να εμφανίσει κάποιο μήνυμα, αν έχει γραφεί κάτι μετά από αυτήν.

Στο παρακάτω παράδειγμα αποφεύγουμε την δημιουργία ενός πολύ μεγάλου πίνακα ακεραίων, τερματίζοντας το πρόγραμμα εάν κάτι τέτοιο ζητηθεί:

```
#define NUM 500
int main( ) {
    #if NUM >= 100000
        #error VERY LARGE ARRAY
    #else
        int pin[NUM];
    #endif
    ..........................................
```

1.2.5. Η οδηγία #undef. Η οδηγία αυτή ακυρώνει τη σταθερά ή τη μακροεντολή, η οποία είχε προηγουμένως οριστεί με μια #define.

Στο παράδειγμα που ακολουθεί, εάν το NUM οριστεί με κάποια τιμή διάφορη του μηδενός, στην οθόνη γράφεται η τιμή του a^2. Εάν το NUM είναι ίσο με μηδέν, η μεταγλώττιση διακόπτεται, ακυρώνεται ο ορισμός του ΤΙΜΗ και εμφανίζεται το μήνυμα λάθους «ΤΙΜΗ NOTPROPERLYDEFINED».

```
#define NUM 5
int  main( ) {
    int a=3;

    #if NUM != 0
        #define TIMH(x) x*x
        printf ("%d\n", TIMH(a));
    #else
        #undef TIMH
        #errorTIMH NOT PROPERLY DEFINED
    #endif
```

1.3. ΠΡΟΚΑΘΟΡΙΣΜΕΝΕΣ ΜΑΚΡΟΕΝΤΟΛΕΣ:

Γράφονται με δύο κάτω παύλες πριν το όνομα και δύο κάτω παύλες μετά το όνομα:

- __**DATE**__ : Παίρνει ως τιμή μια συμβολοσειρά, η οποία περιλαμβάνει την ημερομηνία μεταγλώττισης του προγράμματος. Για

ημερομηνία 19 Οκτωβρίου 2018 για παράδειγμα, η συμβολοσειρά θα είναι η: ”Oct 19 2018”

- **__LINE__** : Παίρνει ως τιμή ένα ακέραιο, όσος ο αριθμός της γραμμής του προγράμματος, στην οποία υπάρχει η σταθερά.
- **__TIME__** : Παίρνει ως τιμή μια συμβολοσειρά, η οποία περιλαμβάνει την ώρα μεταγλώττισης του προγράμματος. Π.χ. ”20:50:18”
- **__FILE__** : Παίρνει ως τιμή μια συμβολοσειρά, η οποία περιλαμβάνει το όνομα του πηγαίου αρχείου.

Για παράδειγμα:

```
printf ("%d\n", __LINE__);
printf ("%s\n", __DATE__);
printf ("%\s", __TIME__);
printf ("%s\n", __FILE__);
```

ΚΕΦΑΛΑΙΟ 2

ΕΙΔΙΚΟΙ ΤΕΛΕΣΤΕΣ ΚΑΙ ΤΥΠΟΙ

2.1. Ο ΤΕΛΕΣΤΗΣ ΚΟΜΜΑ (,).

Χρησιμοποιείται για την ένωση παραστάσεων. Στη συνολική παράσταση η αποτίμηση των επιμέρους παραστάσεων γίνεται από αριστερά προς τα δεξιά. Ο τύπος και η τιμή της συνολικής παράστασης είναι αυτός της τελευταίας επιμέρους παράστασης. Μετά τις παρακάτω εντολές:

```
k = (i=3, j=4, fp=3.6);
printf ("%5d%5d%5.1f%5.1f\n", i, j, fp, k);
x = (i++, j=j-i);

if (i, j)
    printf("EKTYPWSH\n");

printf("%5d%5d", i, j);
```

στην οθόνη θα γραφεί:

```
υυυυ3υυυυ4υυ3.6υυ3.6
υυυυ4υυυυ0
```

Αυτό γιατί, στην μεν πρώτη γραμμή το k παίρνει τιμή όση το fp, ενώ το if της τέταρτης γραμμής δεν εκτελείται, αφού το j έχει γίνει ίσο με μηδέν, άρα η συνθήκη μετά την if είναι ψευδής.

2.2. ΤΕΛΕΣΤΕΣ bit.

Μπορούμε να χειριζόμαστε δεδομένα της C σε επίπεδο bit, δηλαδή να επιδρούμε στα bit από τα οποία αποτελείται ένας ακέραιος αριθμός. Όταν κάνουμε πράξεις σε επίπεδο bit, καλό είναι οι μεταβλητές να δηλώνονται ως unsigned, διαφορετικά πρέπει να λαμβάνουμε υπ' όψη και το bit προσήμου. *Οι τελεστές bit εκτελούν αριθμητικές και όχι λογικές πράξειςμεταξύ των τελεστέων τους, οι οποίοι πρέπει να είναι ακέραιοι τύποι*.

2.2.1. Ο τελεστής &

Εκτελεί την λογική *πράξη AND* μεταξύ των bit δύο αριθμών. Δείτε το αποτέλεσμα της πράξης 38 & 95:

```
0 0 1 0 0 1 1 0    &      (38)
0 1 0 1 1 1 1 1           (95)
------------------
0 0 0 0 0 1 1 0           (6)
```

Οι τελεστές & και && δεν είναι ίδιοι. Για παράδειγμα, το 110 & 17 δίνει αποτέλεσμα μηδέν, ενώ το 110 && 17 δίνει αποτέλεσμα 1.

2.2.2. Ο τελεστής |

Εκτελεί την λογική *πράξη OR* μεταξύ των bit δύο αριθμών. Δείτε το αποτέλεσμα της πράξης 38 & 95:

```
0 0 0 1 0 0 1 1    |      (19)
0 1 1 0 0 0 1 0           (98)
------------------
0 1 1 1 0 0 1 1          (115)
```

Οι τελεστές | και || δεν είναι ίδιοι. Για παράδειγμα, το 19 | 98 δίνει αποτέλεσμα 115, ενώ το 19 || 98 δίνει αποτέλεσμα 1.

2.2.3. Ο τελεστής ^

Εκτελεί την λογική *πράξη XOR* (αποκλειστικό OR) μεταξύ των bit δύο αριθμών. Δείτε το αποτέλεσμα της πράξης 38 & 95:

```
0 0 0 1 0 0 1 1    ^      (19)
0 1 1 0 0 0 1 0           (98)
------------------
0 1 1 1 0 0 0 1          (113)
```

2.2.4. Ο τελεστής ~

Είναι ο *τελεστής συμπληρώματος*, δηλαδή «αντιστρέφει» τα bit ενός αριθμού, με λίγα λόγια, κάνει ίσα με μηδέν όσα bit έχουν τιμή 1 και ίσα με 1 όσα bit έχουν τιμή μηδέν. Έτσι, το ~152 είναι ίσο με 103, αφού:

```
~ 1 0 0 1 1 0 00        (152)
     ισοῦται με:
  0 1 1 0 0 1 1 1       (103)
```

2.2.5. Ο τελεστής <<

Εκτελεί *ολίσθηση προς τα αριστερά* στα bit ενός αριθμού κατά συγκεκριμένο αριθμό θέσεων. Έτσι, η παράσταση a<<b ολισθαίνει προς τα αριστερά κατά b θέσεις τα bit της μεταβλητής a. Στα b χαμηλότερης τάξης bit της a τοποθετεί τιμή μηδέν. Μετά τα παρακάτω:

```
unsigned int a, b=23;
a = b<< 3;
```

το b παραμένει 23, ενώ το a έχει τιμή 184. Το 23 είναι ίσο με το 00010111 στο δυαδικό σύστημα, οπότε η ολίσθησή του αριστερά κατά 3 θέσεις δίνει τον αριθμό 10111000, το οποίο αντιστοιχεί στο δεκαδικό 184. Προφανώς το b<< 4 δίνει τιμή 368 (το πιο αριστερό bit έχει «ανέβει» στο προηγούμενο από τα 4 byte του ακεραίου.

2.2.6. Ο τελεστής >>

Εκτελεί *ολίσθηση προς τα δεξιά* στα bit ενός αριθμού κατά συγκεκριμένο αριθμό θέσεων. Έτσι, η παράσταση a>>b ολισθαίνει προς τα δεξιά κατά b θέσεις τα bit της μεταβλητής a. Στα b υψηλότερης τάξης bit της a τοποθετεί τιμή μηδέν. Μετά τα παρακάτω:

```
unsigned int a, b=178;
a = b>> 3;
```

το b παραμένει 178, ενώ το a έχει τιμή 22. Το 178 είναι ίσο με το 10110010 στο δυαδικό σύστημα, οπότε η ολίσθησή του αριστερά κατά 3 θέσεις δίνει τον αριθμό 00010110, το οποίο αντιστοιχεί στο δεκαδικό 22.

Ας δούμε δυο παραδείγματα χρήσης των παραπάνω τελεστών:

Θα γράψουμε τις εντολές με τις οποίες να διαβάζουμε ένα μη προσημασμένο ακέραιο από 0 έως 255. Στη συνέχεια να αντιμετίθενται οι δύο τετράδες των δυαδικών ψηφίων του και θα εμφανίζεται ο νέος ακέραιος που προκύπτει. *(Από το βιβλίο «C Από τη θεωρία στην Εφαρμογή», Γ.Σ. Τσελίκης, Ν.Δ. Τσελίκας, ISBN: 978-960-93-1961-4).*

```
unsigned int ak, tmp1, tmp2;
scanf("%d", &ak);
tmp1 = ak & 0xF;
tmp1 = tmp1 << 4;
tmp2 = ak >> 4;
ak = tmp1 | tmp2;     ή   ak = tmp1 + tmp2;
printf("%d\n", ak);
```

Δείτε μια πιο πρακτική εφαρμογή των παραπάνω στο παράδειγμα που ακολουθεί *(Από το βιβλίο «Η γλώσσα C σε βάθος Ν. Χατζηγιαννάκης, ISBN: 978-960-461-715-9)*: Έστω ότι μια μετεωρολογική συσκευή μέτρησης, συνδεδεμένη με ένα υπολογιστή, επιστρέφει τόσο την κατάστασή της όσο και τη μέτρηση σε έναν ακέραιο, ως εξής: Το bit 7 του αριθμού έχει τιμή 1 εάν η συσκευή λειτουργεί σωστά. Τα bit 6 και 5 σημαίνουν το είδος της μέτρησης (0 για θερμοκρασία, 1 για πίεση, 2 για υγρασία και 3 για ταχύτητα ανέμου), Τα bit 0 έως και 4

αποτελούν την τιμή της μέτρησης, ενώ τα υπόλοιπα bit (από το 8 έως και το 31) δεν μεταφέρουν χρήσιμες πληροφορίες. Το παρακάτω πρόγραμμα εμφανίζει την κατάσταση της συσκευής, το είδος της μέτρησης και την τιμή της μέτρησης:

```
int main( ) {
   int ak, status, kind, val, ;

   scanf("%d", &ak);
   status = (ak & 128) >> 7;
   kind = (ak & 96) >> 5;
   val = ak & 31;
   if (status == 1)
        printf ("EIDOS = %d. TIMH = %d\n", kind, val);
   else
        printf ("PROBLHMA STHN SYSKEYH\n");
   return 0; }
```

Στο παραπάνω παράδειγμα το ak & 128 απομονώνει το bit 7 του ak και με την εφαρμογή του >>7, το bit αυτό μεταφέρεται στην πρώτη θέση του αριθμού, οπότε αυτός γίνεται 0 ή 1. Αντίστοιχα, με το ak& 96 απομονώνονται τα bit 6 και 5 του ak, ενώ με το ak& 31 απομονώνονται τα πέντε πρώτα bit του ak.

2.3. ΠΡΟΤΕΡΑΙΟΤΗΤΑ ΤΕΛΕΣΤΩΝ.

Η προτεραιότητα τελεστών κατά φθίνουσα σειρά είναι η παρακάτω:

( ) [ ] -> .
& (διεύθυνση)
~ * (για δείκτες) +, - (πρόσημα)
*** / %**
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
= += -= *= /= %= &= |= <<= >>=

2.4. Ο ΤΥΠΟΣ enum.

Είναι ο λεγόμενος *τύπος απαρίθμησης*. Με αυτόν ορίζεται ένα σύνολο από τιμές, τις οποίες μπορεί να πάρει μια μεταβλητή. Έτσιμεταπαρακάτω:

```
enum  month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Noe, Dec};
enum  month  mon;
```

η μεταβλητή mon μπορεί να πάρει μια από τις τιμές που βρίσκονται μέσα στις αγκύλες. Στην πραγματικότητα πάντως, η πρώτη στην σειρά από αυτές τις τιμές παίρνει τιμή 0, η δεύτερη παίρνει τιμή 1 κλπ. Με δεδομένες τις παραπάνω δηλώσεις, εάν δώσουμε την τιμή μηδέν όταν εκτελεστει ο παρακάτω κώδικας, στην οθόνη θα εμφανιστεί το μήνυμα NEW YEAR:

```
scanf("%d", &nm);
if (nm == Jan)
    printf("NEW YEAR\n");
```

Οι αριθμητικές τιμές που αντιστοιχούν στις τιμές που βρίσκονται στις αγκύλες μπορούν να τροποποιηθούν. Έτσι, στο παρακάτω:

```
enum  month  {Jan, Feb, Mar=10, Apr, May, Jun, Jul, Aug=33, Sep, Oct, Noe,
                                                              Dec};
```

το Jan αντιστοιχεί στο 0, το Feb στο 1, το Mar στο 10, το Apr στο 11, το May στο 12, το Jun στο 13, το Jul στο 14, το Aug στο 33, το Sep στο 34, το Oct στο 35, το Noe στο 36 και το Dec στο 37.

Το πρόγραμμα που ακολουθεί θα εμφανίσει στην οθόνη:

```
υυ0υυ1υυ2υυ3υυ4υυ5υυ6υυ7υυ8υυ9υ10υ11
```

```
enum  year  {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
int main( ) {
   int i;
   for (i=Jan; i<=Dec; i++)
          printf("%3d ", i);
   return 0; }
```

αφού όπως προαναφέρθηκε η τιμή του Jan είναι ίση με 0, ενώ του Dec είναι ίση με 11.

ΚΕΦΑΛΑΙΟ 3

ΔΕΙΚΤΕΣ ΚΑΙ ΣΥΝΑΡΤΗΣΕΙΣ

3.1. ΕΙΣΑΓΩΓΙΚΑ ΓΙΑ ΔΕΙΚΤΕΣ – ΣΥΝΤΟΜΗ ΕΠΑΝΑΛΗΨΗ.

α) Η δήλωση μιας μεταβλητής συνεπάγεται την δημιουργία ενός σταθερού δείκτη στη μνήμη, ο οποίος δείχνει την μεταβλητή και έχει ως τιμή την διεύθυνση μνήμης, στην οποία έχει αρχίσει η αποθήκευση της μεταβλητής αυτής. Για παράδειγμα, η δήλωση: **int ak;** συνεπάγεται την δημιουργία του σταθερού δείκτη **&ak**.

β) Η δήλωση ενός πίνακα συνεπάγεται την δημιουργία ενός σταθερού δείκτη στη μνήμη, ο οποίος δείχνει το πρώτο στοιχείο του πίνακα, άρα έχει ως τιμή την διεύθυνση μνήμης, στην οποία έχει αρχίσει η αποθήκευση του πίνακα. Ο δείκτης αυτός έχει το ίδιο όνομα με το όνομα του πίνακα. Για παράδειγμα, η δήλωση:

int mat[N];

συνεπάγεται την δημιουργία του **mat**, ο οποίος είναι σταθερός δείκτης σε ακέραιο.

γ) Η δήλωση: **int *ptr;** δημιουργεί ένα μεταβλητό δείκτη σε ακέραιο. Ο δείκτης αυτός με την δημιουργία του έχει μια άγνωστη τιμή, δείχνει δηλαδή σε άγνωστη θέση μνήμης. Η τιμή **NULL** είναι μια συγκεκριμένη τιμή δείκτη, η οποία είναι καλό να αποδίδεται σε ένα δείκτη με την δημιουργία του. Η τιμή αυτή στην πράξη είναι η μηδενική διεύθυνση μνήμης και μπορεί να αποδοθεί με την εντολή:

ptr = NULL;

Το σύμβολο * στις δηλώσεις του προγράμματος σημαίνει ότι το όνομα που το ακολουθεί προσδιορίζεται ως δείκτης. Μέσα στο πρόγραμμα, το * μπορεί να τεθεί μόνο μπροστά από όνομα δείκτη και σημαίνει «τα περιεχόμενα του δείκτη», ισούται δηλαδή με την τιμή της μεταβλητής στην οποία δείχνει ο δείκτης.

δ) Η τιμή ενός δείκτη μπορεί να εμφανιστεί με την printf( ) και προσδιοριστή %p. Π.χ.:

printf ("%p", ptr);

ε) Εάν το ptr είναι ένας δείκτης σε ακέραιο με τιμή 0021FF22, τότε η τιμή του ptr+1 είναι ίση με 0021FF26. Αυτό, διότι η πρόσθεση εδώ δεν έχει την έννοια της αύξησης κατά μια ακέραια μονάδα, αλλά την αύξηση κατά μια μονάδα αποθήκευσης όπως αυτή στην οποία δείχνει ο ptr. Με δεδομένο ότι ο ptr είναι δείκτης σε ακέραιο, προσθέτοντας 1 στο ptr σημαίνει ένα δείκτη ένα ακέραιο παρακάτω από εκεί όπου δείχνει ο ptr, άρα 4 byte παρακάτω. Δηλαδή δεν γίνεται απλή αριθμητική, αλλά αριθμητική δεικτών.

στ) Στο παρακάτω παράδειγμα, θεωρείστε ότι η αποθήκευση του πίνακα arr έχει αρχίσει από την θέση μνήμης 0028FF10 :

```
int *ptr, arr[5];

ptr = arr;
printf ("%p%p%p%p", ptr, &arr[0], arr, &arr);
```

Η printf θα εμφανίσει το 0028FF10 τέσσερεις φορές. Αυτό διότι:
Το ptr είναι δείκτης σε ακέραιο, ο οποίος έχει τοποθετηθεί στην πρώτη θέση του πίνακα.

Το &arr[0] είναι η διεύθυνση του πρώτου στοιχείου του πίνακα (δείκτης σε ακέραιο).

Το arr είναι και πάλι δείκτης σε ακέραιο, ο οποίος δείχνει την πρώτη θέση του πίνακα.

Το &arr είναι η διεύθυνση όπου βρίσκεται αποθηκευμένος ο δείκτης, δηλαδή τελικά δείκτης σε πίνακα ακεραίων.

3.2. ΔΕΙΚΤΕΣ ΚΑΙ ΠΙΝΑΚΕΣ ΔΥΟ ΔΙΑΣΤΑΣΕΩΝ.

Όπως στους πίνακες μιας διάστασης, έτσι και στους δισδιάστατους, όταν δηλώνουμε πίνακα δημιουργείται και ένας δείκτης, ο οποίος έχει ίδιο όνομα με το όνομα του πίνακα και δείχνει το πρώτο στοιχείο του. Επισημαίνουμε ότι, σε ένα δισδιάστατο πίνακα, το πρώτο στοιχείο του είναι ένας μονοδιάστατος πίνακας. Μετά τις παρακάτω δηλώσεις:

```
int  pin[M][N];
int  (*ptr)[N];
```

το pin είναι πίνακας δύο διαστάσεων MxN, ενώ το ptr είναι δείκτης σε πίνακα N ακεραίων. Δείτε στο παρακάτω παράδειγμα ένα τρόπο για διάβασμα τιμών για τον πίνακα pin και την εμφάνισή τους στην οθόνη με την χρήση του ptr:

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        scanf ("%d", &pin[i][j]);

ptr = pin;

for (i=0; i<M; i++) {
    for (j=0; j<N; j++)
        printf ("%5d", ptr[i][j]);
        printf ("\n"); }
```

Στη συνέχεια θα γράψουμε μια συνάρτηση, την sum( ), η οποία σε ένα μονοδιάστατο πίνακα N θέσεων να γράφει το άθροισμα κάθε στήλης του πίνακα και σε ένα

μονοδιάστατο πίνακα M θέσεων να γράφει το άθροισμα κάθε γραμμής του πίνακα. Τα περιεχόμενα των δύο μονοδιάστατων πινάκων να επιστρέφονται στη main( ). Παρατηρείστε τον ορισμό και τη δήλωση της συνάρτησης, καθώς και το «πέρασμα» του πίνακα pinστην συνάρτηση, με όνομα apin.

```
void sums (int apin[ ][N], int r[M], int c[N]) {
    int j, k;

    for (j=0; j<M; j++)
        r[j] = 0;
    for (k=0; k<N; k++)
        c[k] = 0;

    for (j=0; j<M; j++)
        for (k=0; k<N; k++)
            r[j] += apin[j][k];

    for (j=0; j<M; j++)
        for (k=0; k<N; k++)
            c[k] += apin[j][k];  }
```

Η δήλωση της συνάρτησης αυτής θα είναι:

```
void sums (int [ ][N], int [ ], int [ ]);
```

Ισοδύναμα μπορούμε να έχουμε την παρακάτω δήλωση και επικεφαλλίδα:

```
void sums (int (*apin)[N], intr[M], intc[N])

void sums (int (*)[N], int [  ], int [ ]);
```

Ένας πίνακας συμβολοσειρών είναι επίσης ένας πίνακας δύο διαστάσεων. Στο παρακάτω παράδειγμα διαβάζονται συνεχώς συμβολοσειρές μέχρι να δοθούν M συνολικά συμβολοσειρές ή κενή συμβολοσειρά:

```
char mat[M][N];
int j, k=0;
.....................................
gets (mat[k]);
while (mat[k][0] != '\x0') {
    k++;
    gets(mat[k]); }
```

Συγκριτικά, παρατηρείστε τώρα τις πιο κάτω δηλώσεις και εντολές:

```
int pin[M][N];

int (*ptr)[N];

int (*dkt)[M][N];
```

Με τα for που ακολουθούν γίνεται εισαγωγή στοιχείων στον πίνακα pin.

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        scanf ("%d", &pin[ i ][ j ]);
```

Η εμφάνιση των στοιχείων γίνεται με τη χρήση του pin:

```
for (i=0; i<M; i++) {
     for (j=0; j<N; j++)
          printf ("%5d", pin[ i ][ j ]);
          printf ("\n"); }
```

Με τις παρακάτω εντολές, η εμφάνιση των στοιχείων γίνεται με τη χρήση του ptr:

```
ptr = pin;

for (i=0; i<M; i++) {
     for (j=0; j<N; j++)
          printf ("%5d", ptr[ i ][ j ]);
          printf ("\n"); }
```

Και τέλος, η εμφάνιση των στοιχείων του πίνακα με την χρήση του dkt:

```
dkt = &pin;

for (i=0; i<M; i++) {
     for (j=0; j<N; j++)
          printf ("%5d", (*dkt)[ i ][ j ]);
          printf ("\n"); }
```

Αν η αποθήκευση του pin έχει αρχίσει στην θέση μνήμης 22FE20 και θεωρώντας ότι κάθε δείκτης χρειάζεται 12 byte, τα επόμενα:

```
printf (“%p %p\n”, pin, pin+1);

printf (“%p %p\n”, ptr, ptr+1);

printf (“%p %p\n”, dkt, dkt+1);
```

θα εμφανίσουν προφανώς:

```
22FE20      22FE2C
22FE20      22FE2C
22FE20      22FE38
```


3.3. ΠΙΝΑΚΕΣ ΔΕΙΚΤΩΝ.

Πίνακας δεικτών είναι πίνακας σε κάθε στοιχείο του οποίου υπάρχει ένας δείκτης. Οι δείκτες αυτοί πρέπει να δείχνουν σε ίδιο τύπο δεδομένων (π.χ. πίνακας δεικτών σε ακεραίους, πίνακας δεικτών σε χαρακτήρες κλπ). Ένας πίνακας δεικτών σε ακεραίους N θέσεων, ο pin, θα δηλωθεί ως:

```
int *pin[N];
```

Προσοχή! Η παρακάτω δήλωση δεν είναι το ίδιο:

```
int (*pin)[N];
```

Στην δήλωση αυτή ο pin είναι **δείκτης προς ένα πίνακα Ν ακεραίων**, πρακτικά δηλαδή είναι δείκτης σε δείκτη σε ακέραιο.

Τα στοιχεία ενός πίνακα δεικτών τα χειριζόμαστε με τον ίδιο τρόπο που το κάνουμε και για τους απλούς δείκτες.

Στο παράδειγμα που ακολουθεί γίνεται ταξινόμηση αλφαβητικά κάποιων ονομάτων με την χρήση πίνακα δεικτών σε χαρακτήρα. Κάθε δείκτης του πίνακα έχει τοποθετηθεί να δείχνει την αρχή μιας συμβολοσειράς:

```
char *pin[5], *temp;
int es, ex, i;

pin[0] = "NIKOLAS";
pin[1] = "ANDREAS";
pin[2] = "GIWRGOS";
pin[3] = "KIKITSA";
pin[4] = "EYTERPH";

for (ex=0; ex<4 ; ex++)
     for (es=ex+1 ; es<5 ; es++)
          if (strcmp(pin[ex], pin[es]) > 0){
               temp = pin[es];
               pin[es] = pin[ex];
               pin[ex] = temp; }
for (i=0; i<5; i++)
     puts(pin[i]);
```

Η ταξινόμηση που εφαρμόζεται είναι η λεγόμενη ταξινόμηση με επιλογή

3.4. ΔΕΙΚΤΕΣ ΣΕ ΔΕΙΚΤΕΣ.

Γνωρίζουμε ότι ένας δείκτης ισούται με την διεύθυνση μνήμης μιας μεταβλητής. Με την ίδια λογική, ένας δείκτης μπορεί να ισούται με την διεύθυνση μνήμης ενός άλλου δείκτη, να σημαίνει δηλαδή την θέση μνήμης όπου βρίσκεται αποθηκευμένος ένας δείκτης. Μιλάμε τότε για δείκτη σε δείκτη, στις δηλώσεις δε ενός προγράμματος εμφανίζεται με διπλό αστεράκι. Έτσι, με τις παρακάτω δηλώσεις:

```
int ak=15, *ptr, **dkt;
```

το ptr είναι *δείκτης σε ακέραιο*, ενώ το dkt είναι *δείκτης σε δείκτη σε ακέραιο*. Εάν για παράδειγμα η ακέραια μεταβλητή ak βρίσκεται αποθηκευμένη στην μνήμη στην θέση μνήμης 5000, τότε μετά τις παρακάτω εντολές:

```
ptr = &ak;
dkt = &ptr;
printf ("%p  %p  %p  %d", dkt, *dkt, ptr, ak);
```

θα γραφεί στην οθόνη κάτι τέτοιο:

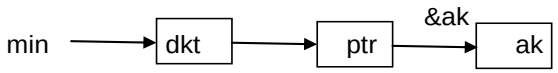
```
6200   5000   5000   15
```

Στο παραπάνω, το ak έχει τιμή 15, το ptr έχει τιμή 5000, αφού έχει γίνει ίση με την θέση στην οποία έχει αποθηκευτεί το ak. Ο dkt είναι ίσος με την θέση στην οποία έχει αποθηκευτεί ο ptr, έστω 6200, άρα η τιμή του dkt είναι ίση με 6200. Τα περιεχόμενα του dkt είναι ίσα με τον ptr, άρα ίσα με 5000.

Με το παρακάτω, η printf( ) θα γράψει στην οθόνη την τιμή 25:

```
int  ak=25, *ptr, **dkt, ***min;
ptr = &ak;
dkt = &ptr;
min = &dkt;
printf ("%d", ***min);
```

Παρακολουθείστε το και σχηματικά:



Μια πολύ συνηθισμένη χρήση δείκτη σε δείκτη είναι όταν μια συνάρτηση πρέπει να μεταβάλει την τιμή ενός δείκτη σε μια άλλη συνάρτηση. Έστω για παράδειγμα ότι σε ένα πρόγραμμα διαβάζεται μια συμβολοσειρά, η pin. Στη συνέχεια καλείται μια συνάρτηση, η οποία θα κάνει τα εξής:

- Θα μετρά τους χαρακτήρες A που υπάρχουν στην συμβολοσειρά (έστω count) και θα επιστρέφει αυτό τον αριθμό.
- Θα ενημερώνει την main( ) για την θέση της συμβολοσειράς όπου βρέθηκε το τελευταίο A (έστω pos η θέση).
- Θα ενημερώνει την main( ) για τον δείκτη που δείχνει στο τελευταίο A (έστω dkt ο δείκτης αυτός).

Στη main( ) θα γράφονται οι τιμές του count, του pos και του dkt.

```
int metr (char*, int*, char**);

int main (void) {
    char pin[N], *dkt;
    int count, pos=0;

    gets (pin);
    count = metr(pin, &pos, &dkt);
    printf ("%5d%5d%15p%15p", count, pos, pin+pos, dkt);
    return 0; }

int metr (char *pin, int *pos, char **dkt) {
    int k, count=0;
    for (k=0; k<strlen(pin); k++) {
        if (pin[k] == 'A') {
            count++;
            *pos = k;
            *dkt = pin+k; } }
        return count; }
```

Παρατηρείστε την παράμετρο dkt στην επικεφαλίδα της συνάρτησης. Έχει τεθεί με διπλό αστεράκι και έτσι μπορεί να ενημερωθεί η τιμή του dkt στη main( ).

3.5. ΔΕΙΚΤΕΣ ΣΕ ΣΥΝΑΡΤΗΣΕΙΣ.

Οι συναρτήσεις, τις οποίες χρησιμοποιεί ένα πρόγραμμα, καταλαμβάνουν χώρο στην μνήμη, όπως άλλωστε και το κύριο πρόγραμμα και οι μεταβλητές. Η διεύθυνση μνήμης από την οποία ξεκινά η αποθήκευση μιας συνάρτησης λέγεται **διεύθυνση της συνάρτησης**. Ένας δείκτης ταυτίζεται στην πραγματικότητα με μια διεύθυνση μνήμης, άρα ένας δείκτης μπορεί να δείχνει σε μια συνάρτηση. Συνεπώς, μια συνάρτηση μπορεί να καλείται με την χρήση των δεικτών που δείχνουν σε αυτήν και όχι με το όνομά της.

Ένας δείκτης σε κάποιου είδους μεταβλητή δείχνει ολόκληρη την μεταβλητή, δηλαδή δεδομένα κάποιου συγκεκριμένου μεγέθους. Ένας δείκτης σε συνάρτηση δεν δείχνει σε δεδομένα κάποιου γνωστού μεγέθους, συνεπώς δεν μπορούμε να κάνουμε αριθμητική δεικτών με δείκτες σε συναρτήσεις.

3.5.1. Συναρτήσεις χωρίς τιμή επιστροφής.

Στο παράδειγμα αυτό, η συνάρτηση display( ) εμφανίζει στην οθόνη την τετραγωνική ρίζα του αριθμού, ο οποίος της δίδεται ως όρισμα. Η συνάρτηση ορίζεται και καλείται σύμφωνα με τον κλασσικό τρόπο:

```
void display (float  fp) {
   printf ("%f", sqrt(fp)); }
int main(void) {
   float  fp;
   scanf ("%f", &fp);
   display (fp);
   return 0; }
```

Χρησιμοποιώντας την έννοια της διεύθυνσης της συνάρτησης, το ίδιο πρόγραμμα μπορεί να γραφεί ως εξής:

```
void display (float  fp) {
   printf ("%f", sqrt(fp)); }
int main(void) {
   float  fp;
   void (*ptr) (float);
   scanf ("%f", &fp);
   ptr = &display;
   (*ptr) (fp);
   return 0; }
```

Σχόλια στο παραπάνω πρόγραμμα:

- Με την δήλωση:

  void (*ptr) (float);

  δηλώνουμε τον δείκτη ptr. Αυτός δείχνει σε μια συνάρτηση με τιμή επιστροφής void και η οποία δέχεται ως όρισμα μια float τιμή.

- Η εντολή:

  ptr = &display;

  τοποθετεί τον δείκτη ptr στη διεύθυνση μνήμης της συνάρτησης display( ). Θυμηθείτε αντίστοιχα πώς θα τοποθετούσαμε π.χ. τον δείκτη σε ακέραιο dkt να δείξει εκεί που είναι αποθηκευμένη η ακέραια μεταβλητή ak. Θα λέγαμε: dkt=&ak;. Η παραπάνω εντολή είναι η πιο συνηθισμένη, όμως ισοδύναμα είναι αποδεκτή και η:

  ptr = display;

- Η εντολή:

  (*ptr) (fp);

  αποτελεί την κλήση της συνάρτησης. Επίσης ισοδύναμα μπορεί να γραφεί:

  ptr (fp);

  Ο πρώτος τρόπος κλήσης είναι προτιμότερος, αφού μας υπενθυμίζει καθαρά ότι το ptr είναι δείκτης σε συνάρτηση και όχι συνάρτηση.

3.5.2. Συναρτήσεις με τιμή επιστροφής.

Στο επόμενο παράδειγμα γίνεται κλήση της συνάρτησης max( ), η οποία έχει δηλωθεί και οριστεί. Η max( ) υπολογίζει και επιστρέφει τον μέγιστο μεταξύ δύο ακεραίων, οι οποίοι της δίνονται ως ορίσματα. Στην περίπτωση αυτή η συνάρτηση δεν είναι προφανώς void:

```
int max ( int, int);

int main(void) {
    int  x, y, meg;
    int (*ptr) (int, int);

    scanf ("%d%d", &x, &y);
    ptr = &max;
    meg = (*ptr) (x, y);
    printf ("%d", meg);
    return 0; }

int max (int x, int y) {
    return (x>=y)? x: y; }
```

3.5.3. Πίνακες δεικτών σε συναρτήσεις.

Σε ένα πίνακα δεικτών σε συναρτήσεις κάθε στοιχείο του είναι δείκτης σε συνάρτηση. Προφανώς οι δείκτες αυτοί είναι όλοι του ίδιου τύπου.

Στο παράδειγμα που ακολουθεί δηλώνεται ένας πίνακας τεσσάρων δεικτών σε συναρτήσεις, ο pin. Οι συναρτήσεις υπολογίζουν το άθροισμα, την διαφορά, το γινόμενο ή το πηλίκο δύο ακεραίων, οι οποίοι διαβάζονται στην main( ). Στη main( ) διαβάζεται επίσης ένας χαρακτήρας (+, -, * ή /), ο οποίος καθορίζει την πράξη που θα γίνει. Αν ο χαρακτήρας δεν είναι ένας από τους παραπάνω, γράφεται η λέξη λάθος και το αποτέλεσμα τίθεται ίσο με μηδέν. Δεν έχει ληφθεί πρόνοια για το τι θα συμβεί στην περίπτωση διαίρεσης με μηδέν.

```
float add (float, float);
float sub (float, float);
float prod (float, float);
float div (float, float);

int main (void) {
    float  x, y, apot=0;
    char ch;
    float (*pin[4]) (float, float);

    pin[0] = &add;
    pin[1] = &sub;
    pin[2] = &prod;
    pin[3] = &div;

    scanf ("%f%f", &x, &y);
    ch = getche( );
    switch (ch) {
        case '+': apot=(*pin[0])(x, y); break;
        case '-': apot=(*pin[1])(x, y); break;
        case '*': apot=(*pin[2])(x, y); break;
        case '/': apot=(*pin[3])(x, y); break;
        default: printf("ERROR\n"); }
    printf ("%f\n", apot);
    return 0; }

float add (float x, float y) {
    return x+y; }
float sub (float x, float y) {
    return x-y; }
float prod (float x, float y) {
    return x*y; }
float div (float x, float y) {
    returnx/y; }
```

Στο παράδειγμα που ακολουθεί χρησιμοποιούμε ένα δισδιάστατο πίνακα ακεραίων, τον arr, στον οποίο δίνουμε τιμές από το πληκτρολόγιο. Ορίζουμε στη συνέχεια μια συνάρτηση, την **incr( )**,η οποία θέλουμε να αυξήσει τα περιεχόμενα κάθε θέσης του πίνακα arr κατά 3 και να επιστρέψει ένα δείκτη στην αρχή του πίνακα (στην πρώτη θέση του). Θα επιστρέψει δηλαδή δείκτη σε πίνακα ακεραίων 4 θέσεων και όχι δείκτη σε ακέραιο, αφού ο arr είναι δισδιάστατος πίνακας. Με την χρήση του δείκτη που επέστρεψε η συνάρτηση, εμφανίζουμε στην οθόνη τα (αυξημένα κατά 3) περιεχόμενα του πίνακα arr:

```
int (*incr (int (*)[4]))[4];

main( ){
     int arr[3][4], j, k;
     int (*dkt)[4];
     for (j=0; j<3; j++)
          for (k=0; k<4; k++)
               scanf ("%d", &arr[j][k]);
     dkt = incr(arr);
     for (j=0; j<3; j++){
          for (k=0; k<4; k++)
               printf ("%5d", dkt[j][k]);
          printf ("\n"); } }

int (*incr (int (*arr)[4]))[4]{
     int j, k;
     for (j=0; j<3; j++)
          for (k=0; k<4; k++)
               arr[j][k] += 3;
     return  arr; }
```

Παρατηρήσεις:

Προφανώς δεν θα μπορούσε να ισχύει η δήλωση:

```
int *incr (int (*)[4]);
```

διότι το dkt είναι δείκτης σε πίνακα ακεραίων 4 θέσεων, όμως σε αυτή την δήλωση η συνάρτηση επιστρέφει δείκτη σε ακέραιο, άρα λάθος.

Η δήλωση:

```
int(*incr) [4] (int (*)[4]);
```

είναι επίσης λάθος, αφού αναφέρεται σε πίνακα συναρτήσεων, σύμφωνα με τα προηγούμενα.

3.6. ΣΥΝΑΡΤΗΣΕΙΣ ΜΕ ΜΕΤΑΒΛΗΤΟ ΑΡΙΘΜΟ ΟΡΙΣΜΑΤΩΝ.

Σε κάποιες περιπτώσεις, ο αριθμός των ορισμάτων με τα οποία καλείται μια συνάρτηση δεν είναι σταθερός. Θυμηθείτε για παράδειγμα τις συναρτήσεις printf( ) και scanf( ).

Όταν ορίζουμε μια συνάρτηση με μεταβλητό αριθμό ορισμάτων, πρέπει να ορίζουμε τουλάχιστον μια υποχρεωτική παράμετρο. Μετά από τις υποχρεωτικές παραμέτρους ακολουθεί μια λίστα με τις μη υποχρεωτικές παραμέτρους, η οποία παριστάνεται με τρείς τελείες.Μια χρήση για παράδειγμα αυτής της παραμέτρου είναι για να καθορίζουμε πόσα είναι τα ορίσματα που ακολουθούν.

Στο παρακάτω παράδειγμα ορίζεται και καλείται η συνάρτηση display( ), η οποία δέχεται μεταβλητό αριθμό ορισμάτων, τα οποία στη συνέχεια εμφανίζει ένα ένα στην οθόνη. Την πρώτη παράμετρο της συνάρτησης (όπως είπαμε και πιο πάνω) θα την χρησιμοποιήσουμε εδώ για να της γνωστοποιήσουμε το πλήθος των μη υποχρεωτικών παραμέτρων:

```
void display (int num,...) {
     va_list user;
     int k, ak;

     va_start (user, num);
     for (k=1; k<=num; k++) {
          ak = va_arg (user, int);
          printf ("%d\n", ak); }
     va_end(user); }


int main( ) {
     display (5, 7, 10, -3, 25, -10); }
```

Σχόλια στο παραπάνω πρόγραμμα:

- Στη συνάρτηση display( ) πρέπει πρώτα-πρώτα να δηλωθεί μια μεταβλητή του τύπου **va_list** (ο οποίος είναι στην πραγματικότητα ένας δείκτης σε χαρακτήρα), η δε προσπέλαση των στοιχείων της λίστας των μη υποχρεωτικών παραμέτρων γίνεται με την χρήση της μεταβλητής αυτής (user).
- Η πρώτη ενέργεια που πρέπει να γίνει είναι η κλήση της συνάρτησης **va_start( )**, η οποία δηλώνεται στο αρχείο **stdarg.h**. Μέσω της va_start( ) κατά κάποιο τρόπο «αρχικοποιείται» η μεταβλητή user. Στη συνάρτηση αυτή το πρώτο όρισμα είναι η μεταβλητή user, ενώ το δεύτερο όρισμα είναι η τελευταία από τις υποχρεωτικές παραμέτρους της συνάρτησης display( ). Μετά την κλήση της va_start( ), ο δείκτης user δείχνει στην πρώτη από τις μη υποχρεωτικές παραμέτρους της display( ). Έτσι, εάν στο πιο πάνω παράδειγμα δώσετε την εντολή **printf ("%d", *user);** μετά την κλήση της va_start( ), στην οθόνη θα εμφανιστεί 7.
- Η συνάρτηση **va_arg( )** δηλώνεται επίσης στο αρχείο stdarg.h. Δέχεται ως παραμέτρους τον δείκτη τύπου va_list (ouser εδώ) και το είδος δεδομένων

μιας από τις μη υποχρεωτικές παραμέτρους της display( ) και επιστρέφει την τιμή της μη υποχρεωτικής παραμέτρου. Μετά την κλήση της va_arg( ), ο user θα δείχνει στην επόμενη μη υποχρεωτική παράμετρο της display( ).

- Η συνάρτηση **va_end( )** δηλώνεται στο stdarg.h και δέχεται ως παράμετρο ένα δείκτη τύπου va_list (τον user εδώ). Με την χρήση της τερματίζεται η προσπέλαση των μη υποχρεωτικών παραμέτρων της display( ).

Οι συναρτήσεις printf( ) και scanf( ), στις οποίες αναφερθήκαμε στην αρχή αυτής της ενότητας, εξάγουν την πληροφορία του είδους των παραμέτρων από τον συντελεστή μορφής που δίνεται.

Άλλο παράδειγμα συνάρτησης με μεταβλητό αριθμό ορισμάτων:

```
void print_str (char *one,...) {
      va_list user;
      char *ptr;

      ptr = one;
      va_start(user, one);
      while (ptr != NULL) {
          puts (ptr);
          ptr = va_arg (user, char*); }
      va_end(user); }
```

Κλήση:

```
print_str("One", "Two", "Three", "Four", NULL);
```

Στην οθόνη θα γραφεί:

One
Two
Three
Four

Αντίστοιχα, αν έχω:

```
char *mat [3] = {"Five", "Six", "Seven"};
.....................................................................
print_str(mat[0], mat[1], mat[2], NULL);
```

θα εμφανιστεί στην οθόνη:

Five
Six
Seven

3.7. ΑΝΑΔΡΟΜΗ.

Αναδρομή είναι η δημιουργία μιας λειτουργίας, κατά την οποία αυτή υλοποιείται με επίκληση των λειτουργιών του εαυτού της. Πρακτικά, μια *αναδρομική συνάρτηση*

είναι μια συνάρτηση η οποία καλεί τον εαυτό της. Απαράβατος κανόνας στις αναδρομικές συναρτήσεις είναι ότι πρέπει να περιέχουν μια συνθήκη τερματισμού (υλοποιείται με μια εντολή ελέγχου), αλλιώς η εκτέλεσή τους θα συνεχίζεται για πάντα (μέχρι να εξαντληθεί η δυνατότητα παροχής μνήμης από τον υπολογιστή).

Στο παρακάτω πρόγραμμα διαβάζεται ο ακέραιος ak και με την κλήση της display( ) εμφανίζονται στην οθόνη οι ακέραιοι από 1 έως τον ak.

```
int main(void) {
   int ak;

   scanf ("%d", &ak);
   display(ak-1); }

void display(int ak) {
   if (ak > 1)
        display(ak-1);
   printf ("%d\n", ak); }
```

Η χρήση αναδρομικών συναρτήσεων είναι συχνή σε μαθηματικά προβλήματα και σε προβλήματα χειρισμού δομών δεδομένων. Λόγω της «ιδιορρυθμίας» στην χρήση αναδρομικών συναρτήσεων, αν μπορούμε να υλοποιήσουμε ένα πρόγραμμα χωρίς την χρήση τους, αλλά με την βοήθεια επαναληπτικών εντολών, προτιμούμε μια τέτοια υλοποίηση. Πάντως με τη χρήση στοίβας, κάθε αναδρομική συνάρτηση μπορεί να μετατραπεί σε μη αναδρομική.

Με την χρήση της παρακάτω συνάρτησης, ένας ακέραιος που δίνεται εμφανίζεται αντεστραμμένος. Δηλαδή ο αριθμός 4536 για παράδειγμα εμφανίζεται ως: 6354

```
void  reverse (intak) {
   if (ak == 0)
        return;
   else
        printf ("%d", ak%10);

   reverse (ak / 10); } }
```

Η επόμενη συνάρτηση υπολογίζει το n!

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

Εδώ υπολογίζουμε τον n-στό αριθμό Fibonacci:

```
int  fibb (int n) {
   if (n == 1 || n == 2)
        return 1;
   return (fibb (n-1) + fibb (n-2)); }
```

ΚΕΦΑΛΑΙΟ 4

ΔΟΜΕΣ (structures) καιΕΝΩΣΕΙΣ (unions)

4.1. ΓΕΝΙΚΑ ΓΙΑ ΤΙΣ ΔΟΜΕΣ -ΥΠΕΝΘΥΜΙΣΕΙΣ.

Χρησιμοποιούνται όταν θέλουμε να λειτουργούμε πάνω σε δεδομένα διαφορετικού τύπου μεταξύ τους, τα οποία να χειριζόμαστε σαν ομάδα. Στην πράξη δηλαδή δημιουργούμε νέα είδη δεδομένων, τα οποία η γλώσσα χειρίζεται όπως και τα κλασσικά και γνωστά σε αυτήν είδη (Int, float κλπ). Υπενθυμίζουμε λοιπόν σύντομα τα εξής:

α) Το παρακάτω αποτελεί την περιγραφή μιας δομής του είδους easy και πρέπει να έχει δοθεί πριν την πρώτη «εμφάνιση» μιας τέτοιου είδους δομής στο πρόγραμμα:

```
struct easy {
   int num;
   char ch; };
```

Με την δήλωση:

```
struct  easy  dom;
```

δημιουργείται η dom, η οποία είναι δομή του είδους easy. Τα ονόματα των πεδίων κάθε δομής του είδους easy που θα υπάρξει στο πρόγραμμα, λέγονται num και ch. Σε αυτά αναφερόμαστε με την χρήση του *τελεστή τελεία* (**.**), π.χ.: **dom.num** και **dom.ch**. Το πλήθος και το είδος των πεδίων μιάς δομής εξαρτάται προφανώς από τις ανάγκες μας. Π.χ.:

```
struct  person {
   char name[30];
   float salary[12];
   int code; };
```

β) Αν οι persa και persb είναι δομές ίδιου τύπου, τότε είναι έγκυρη η εξής απόδοση τιμής:

```
persa = persb;
```

γ) Ένα (ή περισσότερα) πεδία κάποιας δομής μπορεί να είναι δομή κάποιου άλλου τύπου, του οποίου έχει ήδη δοθεί η περιγραφή. Μιλάμε τότε για *εμφωλευμένες δομές*. Π.χ.:

```
struct  atomo {
   char  name[30];
   int code; };

struct couple {
   struct person chief;
   struct person memb;
   float  fp; };
```

Εδώ η προσπέλαση των πεδίων γίνεται με πολλαπλή χρήση του τελεστή τελεία. Έτσι, αν έχουμε την δήλωση:

```
struct couple dok
```

το **dok.fp** είναι ένας float, το **dok.chief.name** είναι πίνακας χαρακτήρων, το **dok.memb.code** είναι ακέραιος κλπ.

δ) Τα πεδία μιας δομής μπορεί να είναι οποιουδήποτε τύπου, άρα και δείκτες, π.χ.:

```
struct diff {
   int     atr;
   char   *name;
   float  *fptr;
   struct couple two; };
```

Η παρακάτω περιγραφή μπορεί για παράδειγμα να χρησιμοποιηθεί στην περιγραφή των κόμβων ενός δυαδικού δέντρου ακεραίων:

```
struct komvos {
   int    data;
   struct komvos *left;
   structkomvos *right; };
```

ε) Όπως ισχύει και για κάθε είδος μεταβλητής, μια συνάρτηση μπορεί να έχει τιμή επιστροφής ένα τύπο δομής, όπως και να δέχεται ως όρισμα δομή.

4.2. ΠΙΝΑΚΕΣ ΔΟΜΩΝ.

Όπως δηλώνουμε πίνακες ακεραίων, πίνακες float κλπ, μπορούμε να δηλώσουμε και πίνακα, κάθε στοιχείο του οποίου είναι μια δομή. Η μεταβλητή pinax που δηλώνεται παρακάτω είναι ένας πίνακας δομών 30 θέσεων, κάθε μια από τις οποίες είναι του τύπου employee:

```
struct employee {
   char name[30];
   int  code;
   float pos; };
.....................................
struct employee pinax[30];
```

Ο χειρισμός των στοιχείων του πίνακα γίνεται όπως και στους γνωστούς απλούς πίνακες δεδομένων. Έτσι, για παράδειγμα οι παρακάτω εντολές είναι έγκυρες:

```
gets (pinax[2].name);

pinax[3] = pinax[5];

strcpy (pinax[2].name, pinax[8].name);

for (k=0; k<30; k++)
     if (pinax[k].name[0] == Ά΄)
          printf("%s\n", pinax[k].name);
```

4.3. ΔΕΙΚΤΕΣ ΚΑΙ ΔΟΜΕΣ.

Όπως προαναφέρθηκε, η προσπέλαση των πεδίων μιας δομής, της οποίας γνωρίζουμε το όνομα, γίνεται με την χρήση του τελεστή τελεία (.). Σε κάποια δομή μπορεί να δείχνει ένας δείκτης σε δομή. Η προσπέλαση των πεδίων της τότε, γίνεται με την χρήση του *τελεστή βέλους* (->).

Στο παράδειγμα που ακολουθεί έχει δηλωθεί μια μεταβλητή δομή, η dok, στην οποία προφανώς δείχνει ο δείκτης &dok. Στην ίδια δομή τοποθετείται να δείχνει και ο δείκτης ptr, ο οποίος έχει δηλωθεί ως δείκτης σε δομές του είδους simple.

```
struct  simple {
    int    num;
    char ch; };

int main(void) {
    struct  simple  dok;
    struct  simple  *ptr;
    ..............................
    ptr = &dok;
    ptr -> num = 303;
    ptr -> ch = 'Q';
    ..............................
```

Με τις δύο τελευταίες εντολές αποδίδονται τιμές στα δυο πεδία της δομής dok. Ως γενική παρατήρηση θυμόμαστε ότι, άσχετα από το πόσα επίπεδα θα συναντήσουμε στην προσπέλαση των πεδίων μιας δομής, *η παράσταση αριστερά από τον τελεστή τελεία (.) πρέπει να είναι όνομα δομής, ενώ αριστερά από τον τελεστή βέλος (->) πρέπει να είναι δείκτης σε δομή*.

4.4. ΠΕΔΙΑ ΣΥΓΚΕΚΡΙΜΕΝΟΥ ΕΥΡΟΥΣ bits (bit fields)

Ας εξετάσουμε αρχικά πόσος χώρος απαιτείται για την αποθήκευση δομών στη μνήμη. Ο χώρος αυτός είναι τουλάχιστον ίσος (και όχι ακριβώς ίσος) με το άθροισμα του χώρου που απαιτείται για το καθένα από τα πεδία της. Το πόσος χώρος ακριβώς απαιτείται εξαρτάται από τον υπολογιστή, μπορούμε όμως προφανώς να τον βρούμε με την χρήση του τελεστή sizeof. Έτσι, για παράδειγμα, για μια δομή του παρακάτω είδους:

```
struct  number {
    int one;
    int two;
    double fp;
    char  ch; };
```

απαιτούνται τόσα byte, όσα δίνει η παράσταση:

```
sizeof (struct  number);
```

Σε κάθε περίπτωση, για να υπολογίσουμε το μέγεθος της μνήμης που απαιτείται, *βρίσκουμε το ελάχιστο ακέραιο πολλαπλάσιο της μεταβλητής που απαιτεί την περισσότερη μνήμη, έτσι ώστε ο αριθμός αυτός να είναι τουλάχιστον ίσος με τον απαιτούμενο χώρο για όλα τα πεδία της δομής*. Έτσι, για μια δομή του είδους number απαιτούνται:

8 byte για το πεδίο fp
4 byte για το πεδίο one
4 byte για το πεδίο two
1 byte για το πεδίο ch

Το ακέραιο πολλαπλάσιο του 8 (όσος ο χώρος για το fp), το οποίο είναι αρκετό για να χωρέσουν όλα τα παραπάνω πεδία, είναι 24, άρα η παραπάνω δομή καταλαμβάνει στη μνήμη χώρο 24 byte.

Μια δομή του παρακάτω είδους, που περιγράφει για παράδειγμα την εγγραφή για ένα φοιτητή

```
struct student {
   char name[40];
   double fp;
   int semest;
   int sex;
   int lessons; };
```

χρειάζεται για την αποθήκευσή της:

40 byte για το πεδίο name
8 byte για το πεδίο fp
4 byte για το πεδίο semest
4 byte για το πεδίο sex
4 byte για το πεδίο lessons

Συνολικά λοιπόν 64 byte.

Πολλές φορές για την αποθήκευση των δομών δεσμεύουμε περισσότερο χώρο στη μνήμη από όσο χρειαζόμαστε. Για παράδειγμα, για ένα πεδίο που μπορεί να πάρει τιμές μόνο 0 ή 1, (χρειάζεται δηλαδή μόνο 1 bit), είμαστε υποχρεωμένοι να δεσμεύσουμε χώρο για ένα ακέραιο, δηλαδή 4 byte. Μπορούμε να καθορίσουμε το πλήθος των bits που καταλαμβάνει κάθε πεδίο της δομής τύπου int ή unsigned (πεδία bit) *θέτοντας την άνω-κάτω τελεία μετά τό όνομα του πεδίου και αμέσως μετά τόν αριθμό των bit, τα οποία θα καταλαμβάνει το πεδίο αυτό*.Τα πεδία bit μεγέθους 1 μπορούν να είναι μόνο unsigned, αφού ένα bit δεν μπορεί να έχει πρόσημο.

Π.χ. για μια δομή που περιέχει το όνομα ενός φοιτητή, το εξάμηνο φοίτησης (8 εξάμηνα συνολικά), το φύλο του (1 ή 0 αγόρι/κορίτσι) και τα μαθήματα που οφείλει (μέχρι 40), η περιγραφή της δομής θα ήταν:

```
struct  student {
        char name [40];
        unsigned  semest: 3;
        unsigned  sex: 1;
        unsigned  lessons: 6; };
```

Η παραπάνω δομή χρειάζεται χώρο 44 byte στη μνήμη, δηλαδή 40 byte για το name και 4 (όσα χρειάζεται ένας ακέραιος) για όλα τα υπόλοιπα πεδία μαζί. Τα τρία τελευταία πεδία καταλαμβάνουν χώρο 10 bit, συνολικά, άρα είναι αρκετός ο χώρος που απαιτείται για ένα ακέραιο. Προφανώς, εάν έλειπε για παράδειγμα το πεδίο lessons, πάλι 44 byte θα καταλάμβανε στη μνήμη. Εάν δεν είχαμε καθορίσει πεδία bit μια δομή του είδους student θα χρειαζόταν προφανώς χώρο 52 byte.

Για την εξοικονόμηση του μεγαλύτερου χώρου στην μνήμη, οι δηλώσεις των πεδίων bit πρέπει να δίνονται όλες μαζί

Η προσπέλαση των πεδίων bit γίνεται όπως και για τα απλά πεδία, όμως αν για παράδειγμα η st είναι δομή του είδους student, *δεν επιτρέπεται* (συντακτικό λάθος) το παρακάτω:

```
scanf ("%d", &st.sex);
```

Επίσης, αν για παράδειγμα δώσω στο πεδίο semest την τιμή 10, αυτό θα πάρει τιμή 10%8==2 (ή άγνωστη τιμή ανάλογα με τον compiler).

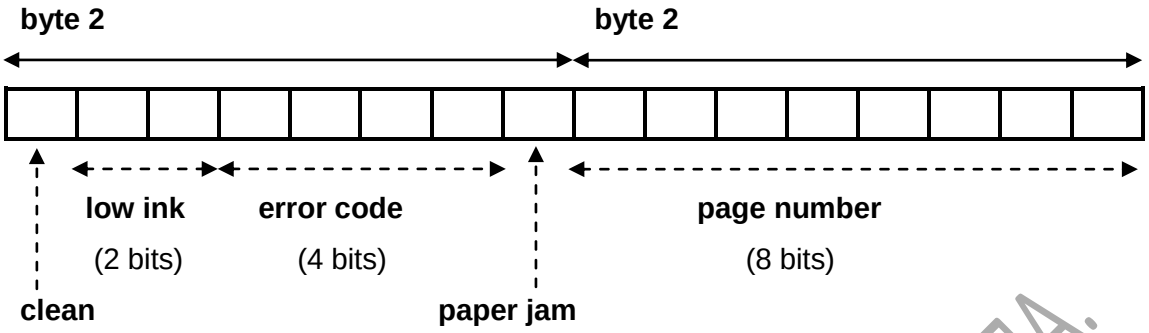
Σχετική άσκηση:

(Από το βιβλίο «C Από τη θεωρία στην Εφαρμογή», Γ.Σ. Τσελίκης, Ν.Δ. Τσελίκας, ISBN: 978-960-93-1961-4).

Το παρακάτω σχήμα 4.1 απεικονίζει τον καταχωρητή κατάστασης (16 bit) ενός εκτυπωτή. Να χρησιμοποιήσετε πεδία bit για να ορίσετε την δομή print_reg με τα πέντε πεδία που εμφανίζονται στο σχήμα. Να γραφεί ένα πρόγραμμα, το οποίο να χρησιμοποιεί αυτή τη δομή για να προσομοιώσει μία εκτυπωτική εργασία 20 σελίδων, ως εξής:

1. Το πεδίο lowink να γίνει ίσο με 3, όταν εκτυπώνεται η $9^{η}$ σελίδα και μέχρι την εκτύπωση της τελευταίας σελίδας.
2. Το πεδίο errorcode να γίνει ίσο με 10, μόνο όταν εκτυπώνεται η $13^{η}$ σελίδα.
3. Το πεδίο paperjam να γίνει ίσο με 1, μόνο όταν εκτυπώνεται η $15^{η}$ σελίδα.
4. Το πεδίο clean να γίνει ίσο με 1, μόνο όταν εκτυπώνεται η τελευταία σελίδα.

Για κάθε σελίδα που εκτυπώνεται, το πρόγραμμα να εμφανίζει την τιμή του καταχωρητή κατάστασης.



Σχ. 4.1

4.5. ΕΝΩΣΕΙΣ (unions).

Είναι παρόμοιες με τις δομές (structures) στο ότι αποτελούνται από ένα ή περισσότερα πεδία διαφόρων τύπων. Οι δομές μας επιτρέπουν να χειριζόμαστε ως μια μονάδα ένα πλήθος διαφορετικών μεταβλητών, αποθηκευμένων σε διαφορετικές θέσεις μνήμης. Οι ενώσεις μας επιτρέπουν να χειριζόμαστε τον ίδιο χώρο μνήμης ως ένα πλήθος από διαφορετικές μεταβλητές, δηλαδή ένα κομμάτι μνήμης αντιμετωπίζεται ως μεταβλητή ενός τύπου σε μια περίπτωση και μεταβλητή διαφορετικού τύπου σε άλλη περίπτωση. Έτσι, μόνο ένα από τα μέλη (τα πεδία) της ένωσης μπορεί να χρησιμοποιηθεί κάθε φορά.

Μια ένωση περιγράφεται με τον ίδιο τρόπο που περιγράφεται και μια δομή, με τη διαφορά ότι χρησιμοποιείται η λέξη-κλειδί union, αντί για την λέξη-κλειδί struct. Η δήλωση εξ άλλου μιας μεταβλητής ένωσης γίνεται με αντίστοιχο τρόπο με αυτόν που γίνεται η δήλωση μιας μεταβλητής δομής.

Στο παρακάτω παράδειγμα το μέγεθος των ενώσεων του είδους mixed είναι 8 byte, παρά το ότι περιέχει ένα ακέραιο και ένα double. Αυτό γιατί, οι δυο μεταβλητές καταλαμβάνουν τον ίδιο χώρο στη μνήμη, οπότε τελικά δεσμεύεται ο χώρος που χρειάζεται η μεγαλύτερη από αυτές. Στο παράδειγμα αυτό, στη γραμμή 13, το πρόγραμμα θα εμφανίσει στην οθόνη «σκουπίδια», διότι θα προσπαθήσει να γράψει ένα double ως ακέραιο.

Παράδειγμα:

```
union mixed {
      int   incase;
      double dcase; };
```

```
int main(void) {
    union mixed unmet;
    printf ("MEGETHOS ENOTHTAS = %5d byte\n",
                        sizeof(union mixed));
    unmet.incase = 159;
    printf ("AKERAIA METABLHTH = %5d\n", unmet.incase);
    scanf ("%lf", &unmet.dcase);
    printf ("DOUBLE METABLHTH = %8.2lf\n", unmet.dcase);
    printf ("AKERAIA METABLHTH = %5d\n", unmet.incase);}     /* 13 */
```

Με τις ενώσεις χρησιμοποιείται ο τελεστής -> με τον ίδιο τρόπο που χρησιμοποιείται και με τις δομές.

Συνηθισμένη εφαρμογή των ενώσεων: η εξοικονόμηση μνήμης. (Ακολουθεί ένα παράδειγμα από το βιβλίο «C Από τη θεωρία στην Εφαρμογή», Γ.Σ. Τσελίκης, Ν.Δ. Τσελίκας, ISBN: 978-960-93-1961-4.)

Έστω ότι θέλουμε να αποθηκεύσουμε σε ένα πίνακα 100 δομών τις προτιμήσεις ανδρών και γυναικών. Οι προτιμήσεις για τους άνδρες περιλαμβάνουν το αγαπημένο τους παιχνίδι και την αγαπημένη τους ταινία, ενώ για τις γυναίκες περιλαμβάνουν την αγαπημένη τηλεοπτική εκπομπή και το αγαπημένο βιβλίο. Για να αποθηκεύσουμε αυτές τις προτιμήσεις, θα μπορούσαμε να χρησιμοποιήσουμε δομές του πιο κάτω είδους:

```
struct  person {
    char  game[20];
    char  movie[30];
    char  show[30];
    char  book[30]; };
```

και να δημιουργήσουμε ένα πίνακα τέτοιων δομών, όπου σε κάθε πεδίο θα αποθηκεύουμε τις προτιμήσεις των ανδρών και των γυναικών. Αυτό θα σήμαινε αφ' ενός μεν σπατάλη μνήμης, αφ' ετέρου δε σχετικά προβλήματα για τα πεδία που δεν χρησιμοποιούνται σε κάθε στοιχείο του πίνακα δομών.

Εναλλακτικά θα μπορούσαμε να υιοθετήσουμε την πιο κάτω προσέγγιση:

```
#define N 100

struct man{
    char game[20];
    char movie[30];};

struct woman{
    char show[30];
    char book[30];};

union data{
    struct man m;
    struct woman w; };

struct person{
    int gn;
    union data d;};
```

```
int main(void) {
    struct person pin[N];
    int k, fylo;

    for (k=0; k<N; k++){

        scanf("%d", &fylo);

        if (fylo == 0)
            pin[k].gn = 0;

        if (fylo == 1)
            pin[k].gn = 1;

        if (fylo == 0){
            printf("Give game: ");
            scanf("%s", pin[k].d.m.game);
            printf("Give movie: ");
            scanf("%s", pin[k].d.m.movie);}

        if (fylo == 1){
            printf("Give show: ");
            scanf("%s", pin[k].d.w.show);
            printf("Give book: ");
            scanf("%s", pin[k].d.w.book); }}

    for (k=0; k<N; k++) {
        if (pin[k].gn == 0)
            printf("Man.  Game:  %s.  Movie:  %s\n",  pin[k].d.m.game,
                    pin[k].d.m.movie);

        if (pin[k].gn == 1)
            printf("Woman.  Show:  %s.  Book:  %s\n",  pin[k].d.w.show,
                    pin[k].d.w.book); } }
```

Στο πρόγραμμα γεμίζει ο πίνακας pin δίνοντας στην μεταβλητή fylo την τιμή 0 εάν πρόκειται για άντρα και 1 εάν πρόκειται για γυναίκα. Στο τέλος εμφανίζονται τα στοιχεία που καταχωρήθηκαν στην οθόνη.

Ως δεύτερη εναλλακτική πρόταση, με μικρές όμως διαφορές, δείτε το παρακάτω:

```
#define SIZE 100
enum gender{MAN, WOMAN};

struct  man{
    char game[20];
    char movie[30];};

struct  woman{
    char show[30];
    char book[30];};

union  data{
    struct man m;
    struct woman w; };
```

```
struct  person{
      enum gender gn;
      union data d;};

int main(void) {
      struct person pin[N];
      int k, fylo;

      for (k=0; k<N; k++){

            scanf("%d", &fylo);

            if (fylo == 0)
                  pin[k].gn = MAN;

            if (fylo == 1)
                  pin[k].gn = WOMAN;

            if (fylo== MAN){
                  printf("Give game: ");
                  scanf("%s", pin[k].d.m.game);
                  printf("Give movie: ");
                  scanf("%s", pin[k].d.m.movie);}

            if (fylo== WOMAN) {
                  printf("Give show: ");
                  scanf("%s", pin[k].d.w.show);
                  printf("Give book: ");
                  scanf("%s", pin[k].d.w.book); }}

      for (k=0; k<N; k++) {
            if (pin[k].gn == MAN)
                  printf("Man.  Game:  %s.  Movie:  %s\n",  pin[k].d.m.game,
                        pin[k].d.m.movie);

            if (pin[k].gn == WOMAN)
                  printf("Woman.  Show:  %s.  Book:  %s\n",  pin[k].d.w.show,
                        pin[k].d.w.book); } }
```

ΚΕΦΑΛΑΙΟ 5

ΔΥΝΑΜΙΚΗ ΔΙΑΧΕΙΡΙΣΗ ΜΝΗΜΗΣ

5.1. ΓΕΝΙΚΑ.

Η C διαχειρίζεται την μνήμη «μοιράζοντάς» την σε τέσσερα τμήματα:

α) Το ένα τμήμα χρησιμοποιείται για την αποθήκευση του κώδικα του προγράμματος.

β) Σε γειτονικό τμήμα μνήμης αποθηκεύονται οι καθολικές και οι εξωτερικές μεταβλητές.

γ) Στο τρίτο τμήμα (*στοίβα, stack*) αποθηκεύονται οι μεταβλητές των συναρτήσεων (αυτόματες μεταβλητές). Η περιοχή αυτή είναι μια δομή τύπου LIFO. Κάθε μεταβλητή που δηλώνεται σε μια συνάρτηση ωθείται στην στοίβα και όποτε περατώνει την εργασία της η συνάρτηση, οι μεταβλητές αυτές ανακαλούνται και καταστρέφονται. Στον ίδιο χώρο αποθηκεύεται η διεύθυνση επιστροφής από τη συνάρτηση. Σημαντικό στοιχείο που πρέπει να θυμόμαστε είναι ότι υπάρχει όριο στο μέγεθος της στοίβας, άρα στο πλήθος και το μέγεθος των μεταβλητών που αποθηκεύονται εκεί. Το όριο εξαρτάται από το λειτουργικό σύστημα. Ο χώρος αυτός μεταβάλλεται, μεγαλώνοντας ή μικραίνοντας ανάλογα με τις μεταβλητές που αποθηκεύονται σε αυτόν, τον διαχειρίζεται δε αυτόματα το λειτουργικό σύστημα.

δ) Το τέταρτο τμήμα μνήμης λέγεται *σωρός* (*heap*). Δεν διατίθεται αυτόματα από το λειτουργικό σύστημα στο πρόγραμμα, αλλά μόνο εάν ο χρήστης το ζητήσει. Αντίστοιχα και η αποδέσμευση του χώρου που έχει παραχωρηθεί είναι αρμοδιότητα του χρήστη και εξαρτάται από αυτόν. Η παραχώρηση μνήμης γίνεται με την χρήση κατά βάση των συναρτήσεων malloc( ) και calloc( ), η δε αποδέσμευση χώρου με την χρήση της συνάρτησης free( ). Σε αντίθεση με τη στοίβα, ο σωρός δεν έχει περιορισμούς ως προς το μέγεθος και το πλήθος των μεταβλητών. Ο μόνος πρακτικός περιορισμός είναι το μέγεθος της μνήμης του υπολογιστή, άρα όποτε απαιτείται η διαχείριση μεγάλης ποσότητας δεδομένων, αυτά είναι καλύτερα να αποθηκεύονται στον σωρό. Η προσπέλαση πάντως του σωρού είναι πιο αργή από την προσπέλαση της στοίβας.

5.2. ΔΕΙΚΤΕΣ void.

Είναι τύπος δεικτών, οι οποίοι δεν σχετίζονται με συγκεκριμένο τύπο δεδομένων, είναι δηλαδή *δείκτες γενικού σκοπού*, θα λέγαμε δηλαδή «δείκτες στα πάντα» ή για να το πούμε διαφορετικά «δείκτες σε τίποτα».

Η δήλωση ενός τέτοιου δείκτη είναι η εξής:

void *ptr;

Ένα δείκτη σε void μπορούμε να τον τοποθετήσουμε έτσι ώστε να δείχνει σε μια μεταβλητή οποιουδήποτε τύπου. Γιαπαράδειγμα:

```
int x;
void *ptr;

ptr = &x;
printf ("%d", *(int *) ptr);
```

Η παραπάνω printf( ) εμφανίζει στην οθόνη την τιμή του x. Δεδομένου ότι ο ptr είναι δείκτης σε void, για να εμφανίσουμε τα περιεχόμενα του πρέπει αναγκαστικά να προηγηθεί προσαρμογή τύπου (casting). Με το:

(int *) ptr

Έχουμε προσαρμογή σε δείκτη σε ακέραιο, τα περιεχόμενα του οποίου (δηλαδή ο ακέραιος, στον οποίο δείχνει) δίνονται από το:

***(int *) ptr**

Στο παρακάτω πρόγραμμα εμφανίζουμε διαδοχικά στην οθόνη την τιμή του ακέραιου x και του floatfp:

```
int x;
float fp;
void *ptr;

ptr = &x;
printf ("%d", *(int *) ptr);
ptr = &fp;
printf ("%f", *(float *) ptr);
```

Στο επόμενο πρόγραμμα:

```
intx=4;
void *ptr=&x;

printf ("%d", *ptr);
printf ("%p", ptr);
printf ("%f", &ptr);
```

η πρώτη printf( ) είναι συντακτικά λανθασμένη, αφού δεν έχει γίνει προσαρμογή τύπου του ptrπριν την εμφάνιση των περιεχομένων του. Οι επόμενες δύο printf( ) εμφανίζουν την τιμή του ptr (διεύθυνση μνήμης) και την διεύθυνση αποθήκευσής του. Δεδομένου ότι δεν είναι γνωστό σε τι δείχνει ένας δείκτης void, δεν επιτρέπεται εν γένει η αριθμητική δεικτών με τέτοιους δείκτες, αν και ορισμένοι compilerστις επιτρέπουν.

5.3. Η ΣΥΝΑΡΤΗΣΗ malloc( ).

Η malloc( ) δέχεται ως όρισμα τον αριθμό των byte των οποίων ζητούμε τη δέσμευση. Στη συνέχεια βρίσκει ένα ελεύθερο μπλοκ μνήμης μεγέθους όσο το πλήθος των byte που ζητήσαμε και επιστρέφει τη διεύθυνση του πρώτου byte αυτού του χώρου. Η τιμή επιστροφής της δηλαδή είναι ένας δείκτης στην αρχή του χώρου που δεσμεύτηκε. Η δήλωσή της είναι:

void *malloc (size_t);

(Το *size_t* είναι ένας μη προσημασμένος (unsigned) ακέραιος τύπος, ο οποίος ορίζεται στο stdlib.h. Χρησιμοποιείται όπου αναφερόμαστε σε αριθμό byte της μνήμης. Μπορεί να σημαίνει ένα οποιοδήποτε μη προσημασμένο τύπο, ανάλογα με την χρήση, δηλαδή μπορεί να σημαίνει unsignedint, unsignedchar, unsignedlong κλπ).

Η malloc( ) έχει *τιμή επιστροφής δείκτη σε void*. Αν υπάρχει αδυναμία δέσμευσης μνήμης, η malloc( ) επιστρέφει ένα μηδενικό δείκτη (NULL). *Κατά την χρήση της οφείλουμε να ελέγχουμε πάντα την δυνατότητα δέσμευσης μνήμης*.

Στο παρακάτω πρόγραμμα δεσμεύεται χώρος για num χαρακτήρες. Το num δίνεται από τον χρήστη. Στον χώρο αυτό καταχωρούνται χαρακτήρες οι οποίοι διαβάζονται από το πληκτρολόγιο μέχρι να δοθούν num συνολικά ή μέχρι να δοθεί τελεία. Στη συνέχεια το πρόγραμμα εμφανίζει τους χαρακτήρες που δόθηκαν:

```
int  main(void) {
   int num, j, k;
   char *str, ch;

   scanf ("%d", &num);
   str = (char *) malloc(num);
   if (str == NULL) {
        printf ("Anavailable memory\n");
        return 0; }
   else {
        for (k=0; k<num; k++) {
             ch = getche ( );
             if (ch=='.')
                 break;
             *(str+k) = ch; }
        printf("\n");

        for (j=0; j<k; j++)
             printf("%c", *(str+j));
        printf("\n");
        return 0; }
```

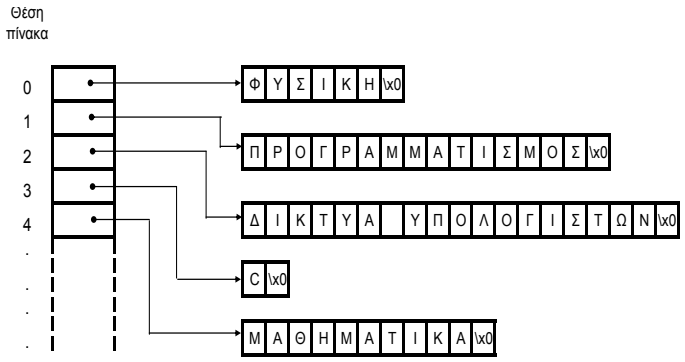
Στο παραπάνω πρόγραμμα γίνεται προσαρμογή τύπου (casting) του δείκτη σε void, τον οποίο επιστρέφει η malloc( ) σε δείκτη σε χαρακτήρα. Αυτή η προσαρμογή, αν

και δεν είναι απαραίτητη σε πολλούς compilers, χρησιμοποιείται για να εξασφαλίσει την μέγιστη δυνατή φορητότητα στο πρόγραμμα.

Στο παράδειγμα που ακολουθεί γίνεται χρήση ενός πίνακα δεικτών σε χαρακτήρες. Με το πρόγραμμα αυτό γίνεται προφανής η εξοικονόμηση χώρου κατά το διάβασμα και την αποθήκευση συμβολοσειρών, σε σχέση με την αποθήκευσή τους σε ένα δηλωμένο (άρα στατικό) πίνακα χαρακτήρων, για τον οποίο η δέσμευση χώρου είναι συγκεκριμένη από την αρχή του προγράμματος. Το πρόγραμμα διαβάζει συνεχώς συμβολοσειρές από το πληκτρολόγιο, 80 χαρακτήρων το πολύ κάθε μια. Το διάβασμα συνεχίζεται μέχρι να διαβαστούν συνολικά 100 συμβολοσειρές ή μέχρι να δοθεί μια μηδενική συμβολοσειρά, δηλαδή κάποια με τον χαρακτήρα '\x0' ως πρώτο στοιχείο της. Οι συμβολοσειρές που διαβάζονται υποτίθεται ότι είναι τίτλοι βιβλίων. Δημιουργείται έτσι ένας κατάλογος βιβλίων, τα περιεχόμενα του οποίου εμφανίζονται στην οθόνη ένα προς ένα στο τέλος του προγράμματος.

```
#define LINE 81
#define MAX 100

int main(void) {
    char temp[LINE], ch;
    char *ps[MAX];
    int index=0, k;

    puts("TITLES OF BOOKS");
    gets(temp);
    while ((index < MAX) && (temp[0] != '\x0')) {
        ps[index] = (char *) malloc(strlen(temp)+1);
        strcpy (ps[index], temp);
        index++;
        gets(temp); }

    putchar('\n');
    puts("LIST OF BOOKS");
    for (k=0; k<index; k++)
        puts(ps[k]); }
```

Είναι προφανές ότι οι δείκτες του πίνακα ps δείχνουν σε περιοχές μνήμης, στην αρχή συμβολοσειρών, οι οποίες δεν είναι κατ' ανάγκην του ίδιου μεγέθους όλες. Έτσι, αν για παράδειγμα οι πέντε πρώτοι τίτλοι βιβλίων που διαβάστηκαν είναι: ΦΥΣΙΚΗ, ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ, ΔΙΚΤΥΑ ΥΠΟΛΟΓΙΣΤΩΝ, C, ΜΑΘΗΜΑΤΙΚΑ, μετά την εκτέλεση του προγράμματος στη μνήμη θα έχουμε την παρακάτω κατάσταση (σχ 5.1):



Σχ. 5.1

5.4. Η ΣΥΝΑΡΤΗΣΗ calloc( ).

Η συνάρτηση calloc( ) δεσμεύει επίσης μνήμη την ώρα της εκτέλεσης ενός προγράμματος. Δέχεται ως ορίσματα τον αριθμό των στοιχείων μνήμης που θα δεσμευτούν, καθώς και το πλήθος των byte ανά στοιχείο μνήμης. Δεσμεύει δηλαδή τόσα byte, όσο το γινόμενο των δύο ορισμάτων της. Η δήλωσή της είναι:

void *calloc (size_t, size_t);

Εάν η δέσμευση χώρου είναι επιτυχής, *μηδενίζει τα περιεχόμενα της μνήμης που δεσμεύει* και επιστρέφει ένα δείκτη στην αρχή αυτού του χώρου. Όπως και στην malloc( ), η τιμή επιστροφής της είναι δείκτης σε void, οπότε είναι αναγκαία η προσαρμογή τύπου. Σε αδυναμία δέσμευσης μνήμης επιστρέφει ένα μηδενικό δείκτη (NULL). Όπως και και στην malloc( ), *κατά την χρήση της οφείλουμε να ελέγχουμε πάντα την διαθεσιμότητα μνήμης*, την δυνατότητα δηλαδή δέσμευσης μνήμης.

Στο πιο κάτω παράδειγμα, η calloc( ) δεσμεύει χώρο για να τοποθετηθούν num ακέραιοι. Η δεύτερη for εμφανίζει τα περιεχόμενα του χώρου που δεσμεύτηκε στην οθόνη.

```
int main (void) {
    unsigned num;
    int *ptr, k, ak;

    scanf ("%d", &num);
    ptr = (int*) calloc (num, sizeof (int));
    if (ptr == NULL) {
        printf ("Anavailable memory\n");
        return 0; }
    else {
        for (k=0; k<num; k++) {
            scanf("%d", &ak);
            *(ptr+k) = ak; }
    printf("\n");

    for (k=0; k<num; k++)
        printf("%5d\n", *(ptr+k));
    return 0; }
```

5.5. Η ΣΥΝΑΡΤΗΣΗ realloc( ).

Η συνάρτηση realloc( ) *τροποποιεί την ποσότητα μνήμης που είχε προηγουμένως δεσμευτεί* από κλήση είτε της malloc( ) είτε της calloc( ).Η δήλωσή της είναι:

void *realloc (void *, size_t);

Το πρώτο της όρισμα είναι ένας δείκτης στη θέση μνήμης, της οποιας το μέγεθος θέλουμε να τροποποιήσουμε και την οποία έχουμε προηγουμένως δεσμεύσει δυναμικά.

Το δεύτερό της όρισμα είναι το νέο πλήθος των byte που θα δεσμευτούν. Το μέγεθος της νέας θέσης στη μνήμη πρέπει να υπολογιστεί πριν από την κλήση της realloc( ).

Ανάλογα με τη διαθεσιμότητα της μνήμης, το αποτέλεσμα της realloc( ) είναι ένα από τα επόμενα:

- Αν τα byte των οποίων ζητείται η δέσμευση είναι λιγότερα από τα ήδη δεσμευμένα, τότε η realloc( ) απελευθερώνει όσο χώρο δεν χρειάζεται πλέον.
- Αν τα byte των οποίων ζητείται η δέσμευση είναι περισσότερα από τα ήδη δεσμευμένα, η realloc( ) ίσως να μην είναι σε θέση να επεκτείνει το ήδη δεσμευμένο τμήμα μνήμης. Τότε δεσμεύει χώρο σε νέα θέση, τα δε υπάρχοντα δεδομένα αντιγράφονται στη νέα θέση. Το παλιό μπλόκ μνήμης απελευθερώνεται και η συνάρτηση επιστρέφει ένα δείκτη στην αρχή του νέου μπλόκ.
- Σε αδυναμία δέσμευσης μνήμης επιστρέφει ένα μηδενικό δείκτη (NULL). Αν ο χώρος δεσμευτεί επιτυχώς, επιστρέφει δείκτη στην αρχή του χώρου αυτού.

Η realloc( ) χρειάζεται προσαρμογή τύπου όπως η calloc( ) και η malloc( ).

Στο παράδειγμα που ακολουθεί γίνεται χρήση της realloc( ), προκειμένου να επεκταθεί ο χώρος που δεσμεύτηκε αρχικά για την αποθήκευση μιας συμβολοσειράς.

```
int main(void) {
        char buf[80], *mes;
        scanf ("%s", buf);
        mes = (char *) malloc(strlen(buf)+1);
        strcpy(mes,buf);
        puts("Enter another line");
        gets(buf);
        mes = (char *) realloc(mes, (strlen(mes)+strlen(buf)+1));
        strcat(mes,buf);
        puts(mes);
        return 0; }
```

5.6. Η ΣΥΝΑΡΤΗΣΗ free( ).

Η free( ) *ακυρώνει τη δέσμευση μνήμης* που έγινε προηγουμένως με την χρήση της calloc( ), της malloc( ) ή της realloc( ). Η free( ) δέχεται ως όρισμα ένα δείκτη στην περιοχή μνήμης που πρόκειται να αποδεσμευτεί. Έτσι, στο δεύτερο παράδειγμα της παραπάνω παραγράφου 5.2, η εντολή:

free (ps[2]);

θα απελευθέρωνε την περιοχή μνήμης που αρχίζει στη διεύθυνση που καθορίζεται από το ps[2]. Σαν περιεχόμενο του ps[2] παραμένει η ίδια διεύθυνση, όμως δεν έχουμε τη δυνατότητα να αποθηκεύσουμε εκεί πληροφορία, αφού η μνήμη έχει απελευθερωθεί. Το ps[2] μπορεί να χρησιμοποιηθεί ξανά στη συνέχεια αν καλέσουμε την calloc( ) ή τη malloc( ) και αποθηκεύσουμε τον δείκτη που επιστρέφεται από αυτήν στο ps[2].

Η διαχείριση της μνήμης πρέπει να γίνεται με κριτήριο την οικονομία σε ένα πρόγραμμα. Από ανεξέλεγκτη χρήση της μνήμης μπορούν εύκολα να προκύψουν προβλήματα, υπερχειλίσεις κλπ. Έτσι, μνήμη που δεν χρησιμοποιείται είναι αναγκαίο να αποδεσμεύεται.

Στο πρόγραμμα που ακολουθεί, η test( ) εμφανίζει example, ενώ αμέσως μετά απελευθερώνεται ο χώρος στον οποίο έδειχναν τόσο op1, όσο και ο p2, οπότε η printf( ) της main( ) εμφανίζει «σκουπίδια».

(Από το βιβλίο «C Από τη θεωρία στην Εφαρμογή», Γ.Σ. Τσελίκης, Ν.Δ. Τσελίκας, ISBN: 978-960-93-1961-4.)

```
intmain(){
   char *p1, *p2;

   p1 = p2 = (char *) malloc(8);
   if (p1 != NULL){
        strcpy(p1, "example");
        test(p2);
        printf("%s\n", p1); }
   return 0;}

void test(char *p){
   printf("%s\n", p);
   free(p); }
```

Στο παρακάτω πρόγραμμα η απελευθέρωση του p δεν σημαίνει ότι κάνει και τον δείκτη p ίσο με NULL, οπότε η δεύτερη printf( ) θα βγάλει «σκουπίδια» στην οθόνη.

```
int main ( ) {
   char *p;


   p = (char *) malloc(8);
   if (p != NULL) {
        strcpy (p, "example");
        printf ("%s\n", p); }
   free (p);
   if (p != NULL)
        printf("%s\n", p);
   return 0; }
```

5.6. ΔΗΜΙΟΥΡΓΙΑ ΔΥΝΑΜΙΚΩΝ ΠΙΝΑΚΩΝ.

Με τη χρήση των συναρτήσεων δυναμικής δέσμευσης μνήμης, μπορούμε να δημιουργήσουμε *«δυναμικούς πίνακες»*. Με αυτό εννοούμε «πίνακες», των οποίων το μέγεθος δεν είναι γνωστό εκ των προτέρων, δεν έχει δηλαδή δηλωθεί στον χώρο δήλωσης μεταβλητών. Είναι προφανές ότι ο όρος «πίνακας» χρησιμοποιείται εδώ καταχρηστικά, σε σχέση με όσα ήδη γνωρίζουμε, αφού ο πίνακας είναι στατική δομή, δηλαδή ο χώρος που καταλαμβάνεται γι'αυτόν είναι δεδομένος από την αρχή του προγράμματος και μέχρι την λήξη του.

5.6.1. Δυναμικοί μονοδιάστατοι πίνακες.

Στο πρώτο παράδειγμα της παραγράφου 5.2. έγινε δυναμική δέσμευση μνήμης για ένα σύνολο από χαρακτήρες. Παρατηρείστε ότι τον χώρο που δεσμεύτηκε πρακτικά από την δέσμευσή του και μετά τον « βλέπουμε» σαν πίνακα num θέσεων. Στην αρχή του χώρου αυτού δείχνει ένας δείκτης, ο str, η δε καταχώρηση στοιχείων γίνεται με την χρήση επαναληπτικής εντολής, όπως παρακάτω, χρησιμοποιώντας συμβολισμό δεικτών:

```
for (k=0; k<num; k++) {
     ch = getche ( );
     *(str+k) = ch; }
```

Με αυτό τον τρόπο δηλαδή δημιουργήσαμε ένα δυναμικό «πίνακα», του οποίου το πλήθος των θέσεων καθορίζεται από τον χρήστη. Αντίστοιχα, στο παράδειγμα που ακολουθεί γίνεται δέσμευση χώρου για n ακέραιους, δημιουργώντας έτσι ένα δυναμικό πίνακα ακεραίων n θέσεων. Τον χώρο αυτό

τον χειριζόμαστε και εδώ ως πίνακα, χρησιμοποιώντας αυτή τη φορά συμβολισμό πινάκων:

```
int k, n, num, *pin;
.....................................
scanf ("%d", &n);
pin = (int *) malloc (n*sizeof(int));
for (k=0; k<n; k++) {
      printf("pin[%d]: \n", k);
      scanf("%d", &num);
      pin[k]=num;}
```

5.6.2. Δυναμικοί δισδιάστατοι πίνακες.

Στο επόμενο παράδειγμα γίνεται δυναμική δημιουργία ενός δισδιάστατου πίνακα ακεραίων nxm:

```
int main(void) {
      int k, j, n, m,  num, **mat;

      scanf ("%d%d", &n, &m);
      mat = (int **)malloc(n*sizeof(int *));
      for (k=0; k<n; k++)
            mat[k] = (int *) malloc(m*sizeof(int));
      for(j=0; j<n; j++) {
            for(k=0; k<m; k++) {
                  printf("mat[%d][%d]: \n", j, k);
                  scanf("%d", &num);
                  mat[ j ][k]=num; } } }
```

Η πρώτη malloc στο πρόγραμμα αυτό δεσμεύει χώρο για n δείκτες σε ακέραιο, πρακτικά δηλαδή κατασκευάζει ένα πίνακα n δεικτών. Με την πρώτη for δεσμεύεται για n φορές χώρος m ακεραίων και καθένας από τους n δείκτες που δημιουργήθηκαν προηγουμένως, τοποθετείται να δείχνει σε ένα από τους n αυτούς χώρους. Παρατηρείστε ότι ο mat δηλώνεται ως δείκτης σε δείκτη σε ακέραιο:

```
int **mat;
```

ΚΕΦΑΛΑΙΟ 6

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

6.1. ΓΕΝΙΚΑ.

Λέγοντας «Δομές Δεδομένων» εννοούμε τρόπους με τους οποίους οργανώνονται τα χρησιμοποιούμενα δεδομένων στην μνήμη (κύρια ή δευτερεύουσα) του υπολογιστή. Μιλούμε έτσι κυρίως για πίνακες, στοίβες, ουρές, λίστες, δέντρα και γράφους. Θα εξετάσουμε κάποιες από αυτές με την χρήση της C, εκμεταλλευόμενοι το «εργαλείο» των δεικτών, το οποίο μας παρέχει η γλώσσα και το οποίο κάνει την υλοποίηση των δομών αρκετά εύκολη και αποδοτική. Οι πίνακες, οι στοίβες, οι ουρές και οι λίστες είναι *γραμμικές δομές*, δηλαδή δομές στις οποίες κάθε στοιχείο έχει ένα προηγούμενο και ένα επόμενο. Δεν θα αναφερθούμε σε πίνακες, σημαντικά θέματα των οποίων έχουμε ήδη εξετάσει. Θα αναφερθούμε επίσης σε στοίβες και ουρές μόνο μέσω παραδειγμάτων. Υπενθυμίζουμε ότι:

- *Στοίβα* είναι μια δομή δεδομένων, η οποία υπακούει στον κανόνα LIFO (LastInFirstOut).
- *Ουρά* είναι μια δομή δεδομένων, η οποία υπακούει στον κανόνα FIFO (FirstInFirstOut).

6.2. ΣΥΝΔΕΔΕΜΕΝΕΣ ΛΙΣΤΕΣ.

Είναι γραμμικές δομές, στις οποίες η λογική σειρά των στοιχείων στη μνήμη δεν συμβαδίζει απαραίτητα με φυσική σειρά. Αποτελούνται από μια σειρά *κόμβων*, των οποίων το πλήθος δεν είναι εν γένει γνωστό εκ των προτέρων, καθένας δε από αυτού περιλαμβάνει «χρήσιμα» δεδομένα και δείκτη ή δείκτες προς άλλο ή άλλους κόμβους. Έτσι, διακρίνονται σε *απλά* και *διπλά* συνδεδεμένες λίστες. Ως κύριες λειτουργίες στις λίστες μπορεί κανείς να θεωρήσει τις εξής:

- Εισαγωγή στοιχείου (στην αρχή ή στο τέλος ή σε τυχαία θέση της λίστας).
- Διαγραφή στοιχείου (από την αρχή ή το τέλος ή από τυχαία θέση της λίστας).
- Αναζήτηση στοιχείου.
- Συνένωση λιστών.
- Αντιστροφή λίστας.
- Μετακίνηση κόμβου μέσα στην λίστα.
- Διαγραφή λίστας.

Με την χρήση λίστας μπορούν να υλοποιηθούν με δυναμικό τρόπο άλλες δομές δεδομένων, όπως για παράδειγμα στοίβες, ουρές κλπ.

6.2.1. Απλά συνδεδεμένες λίστες.

Σε κάθε κόμβο της λίστας υπάρχουν «χρήσιμα» δεδομένα και ένας δείκτης προς τον λογικά επόμενο κόμβο. Η περιγραφή για παράδειγμα των κόμβων μιας απλά συνδεδεμένης λίστας ακεραίων θα δίδεται από την παρακάτω δομή:

```
struct node {
    int data;
    struct node *next; };
```

Προκειμένου να υπάρχει η δυνατότητα διάσχισης της λίστας, πρέπει οπωσδήποτε να υπάρχει ένας δείκτης ο οποίος να δείχνει στον πρώτο κόμβο της (κεφαλή της λίστας) και ο οποίος να μη μετακινηθεί ποτέ από τον κόμβο αυτό. Ο δείκτης εξ άλλου του τελευταίου κόμβου της λίστας πρέπει να τερματίζεται (να δείχνει) σε NULL. Με την χρήση των δύο αυτών δεικτών και αφού τοποθετήσουμε ένα βοηθητικό δείκτη στην κεφαλή της λίστας μπορούμε να διατρέξουμε ολόκληρη την λίστα. Έτσι, αν ο δείκτης head είναι ο μη μετακινούμενος δείκτης στην κεφαλή της λίστας, με τις παρακάτω εντολές διασχίζουμε την λίστα και γράφουμε το ακέραιο πεδίο κάθε κόμβου:

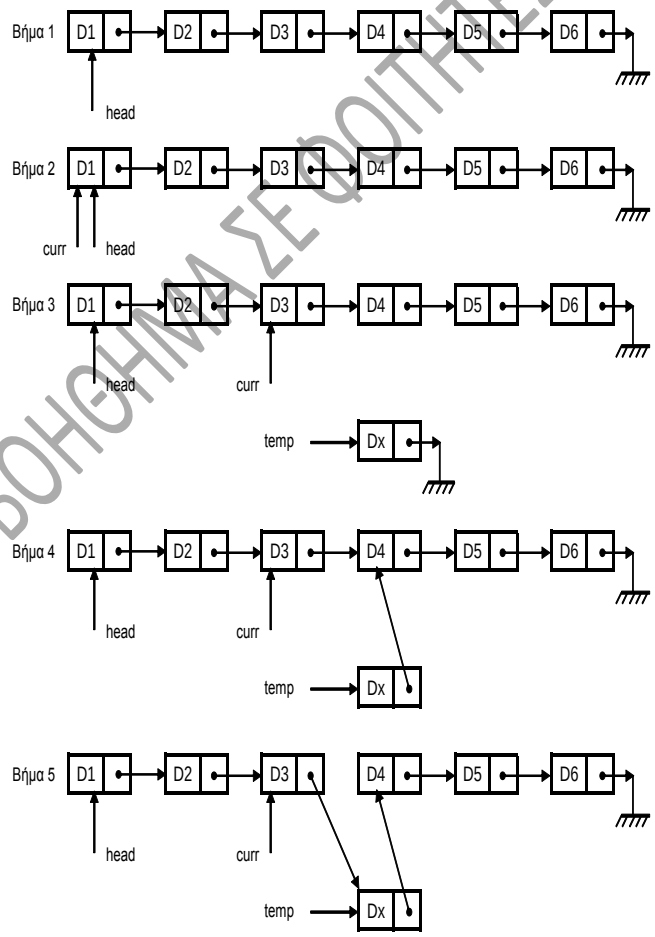
```
struct node *head, *curr;
.....
curr = head;
while (curr != NULL) {
     printf("%d", curr -> data);
     curr = curr -> next; }
```

Προσέξτε ότι η εντολή curr = curr + 1; δεν είναι ισοδύναμη με την τελευταία εντολή της παραπάνω επανάληψης, αφού οι κόμβοι της λίστας δεν βρίσκονται κατ' ανάγκη αποθηκευμένοι σε διαδοχικές θέσεις μνήμης.

Στο παρακάτω παράδειγμα παρουσιάζεται ενδεικτικά μια συνάρτηση εισαγωγής ενός κόμβου σε συγκεκριμένη θέση μιας απλά συνδεδεμένης λίστας. Η συνάρτηση δέχεται ως ορίσματα τον δείκτη head στην κεφαλή της λίστας και ένα ακέραιο, τον pos, ο οποίος ισούται με τον αύξοντα αριθμό του κόμβου, μετά από τον οποίο θα γίνει η εισαγωγή. Οι κόμβοι της λίστας είναι του τύπου node, όπως περιγράφεται παραπάνω:

```
void insert_list (struct node *head, int pos) {
    int  k, ak;
    struct node *temp, *curr;

    curr = head;
    for (k=1; k<pos; k++)
        curr = curr -> next;
    temp = (struct node *) malloc (sizeof (struct node));
    if (temp != NULL) {
        temp -> next = NULL;
        scanf ("%d",&temp -> data);
        temp -> next = curr -> next;
        curr ->next = temp; }}
```

Στο σχήμα 6.1 που ακολουθεί φαίνεται η σειρά των ενεργειών στις οποίες προβαίνουμε για την εισαγωγή κόμβου, σύμφωνα με την παραπάνω συνάρτηση. Θεωρούμε ότι η εισαγωγή του νέου κόμβου θα γίνει μετά τον τρίτο κόμβο της λίστας:



Σχ. 6.1

Η παρακάτω συνάρτηση αντιστρέφει μια απλά συνδεδεμένη λίστα και επιστρέφει ένα δείκτη στην νέα κεφαλή της λίστας:

```
struct node *reverse (struct node *head) {
    struct node *a, *b;

    a = head -> next;
    b = a -> next;
    head -> next = NULL;
    while (a -> next != NULL) {
        a -> next = head;
        head = a;
        a = b;
        b = b -> next; }
    a -> next = head;
    head = a;
    a = NULL;
    returnhead; }
```

Ησυνάρτηση destroy( ) που ακολουθεί διαγράφει μια υπάρχουσα λίστα:

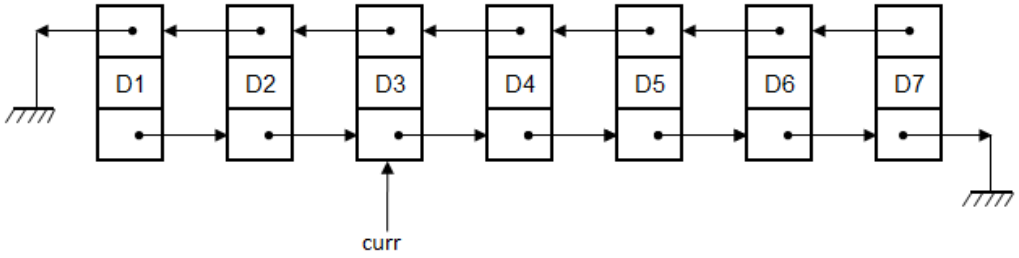
```
void destroy (struct node *head) {
    struct node *ptr;

    ptr = head;
    while (ptr != NULL) {
        head = head -> next;
        free (ptr);
        ptr = head; }}
```

6.2.2. Διπλά συνδεδεμένες λίστες.

Σε κάθε κόμβο της λίστας υπάρχουν «χρήσιμα» δεδομένα και δύο δείκτες. Ο ένας δείχνει προς τον λογικά επόμενο κόμβο και ο άλλος δείχνει προς τον λογικά προηγούμενο κόμβο. Η περιγραφή για παράδειγμα των κόμβων μιας διπλά συνδεδεμένης λίστας ακεραίων θα δίδεται από την παρακάτω δομή:

```
struct dnode {
    int data;
    struct dnode *left;
    structdnode *right; };
```

Σε μια διπλά συνδεδεμένη λίστα δεν έχει νόημα να μιλήσουμε για «κεφαλή», αφού η διάσχιση της λίστας μπορεί να γίνει και προς τις δύο «κατευθύνσεις». Προκειμένου να υπάρχει η δυνατότητα διάσχισης της λίστας, πρέπει οπωσδήποτε να είναι γνωστός ένας δείκτης (έστω curr), ο οποίος να δείχνει σε κάποιο κόμβο της. Εδώ ο τερματισμός της λίστας γίνεται και προς τις δύο κατευθύνσεις με τους δείκτες των δύο ακραίων κόμβων να δείχνουν σε NULL.Παραστατικά μια διπλά συνδεδεμένη λίστα παρουσιάζεται στο σχ. 6.2.



Σχ. 6.2

Με τις παρακάτω εντολές αρχικά μετακινούμε τον δείκτη curr στο ένα άκρο της λίστας και στην συνέχεια την διασχίζουμε και γράφουμε το ακέραιο πεδίο κάθε κόμβου στην οθόνη, κινούμενοι προς την αντίθετη «κατεύθυνση»:

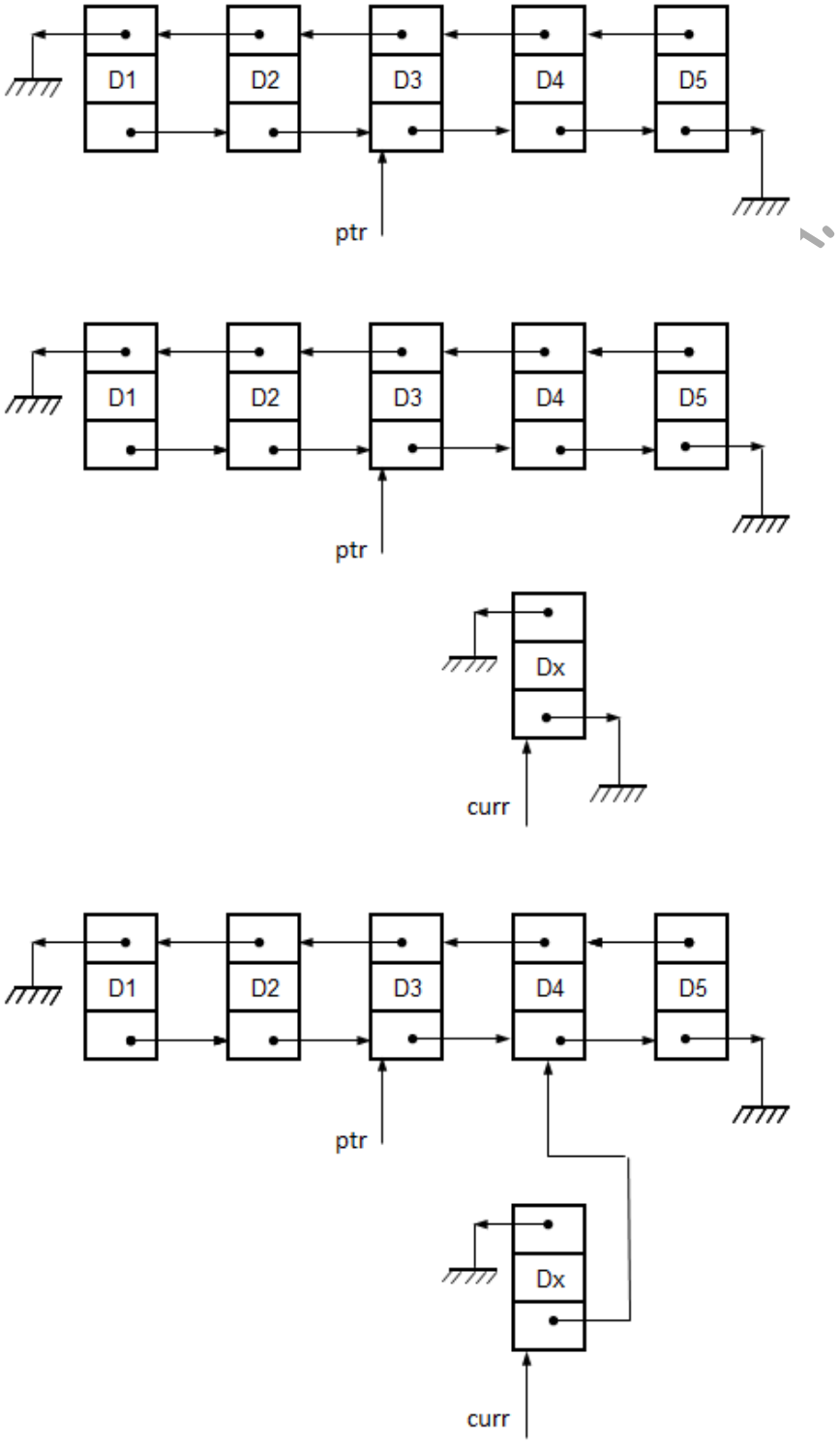
```
struct dnode *curr;
.....
while (curr -> left != NULL)
     curr = curr ->left;
while (curr != NULL) {
     printf("%d", curr -> data);
     curr = curr ->right; }
```

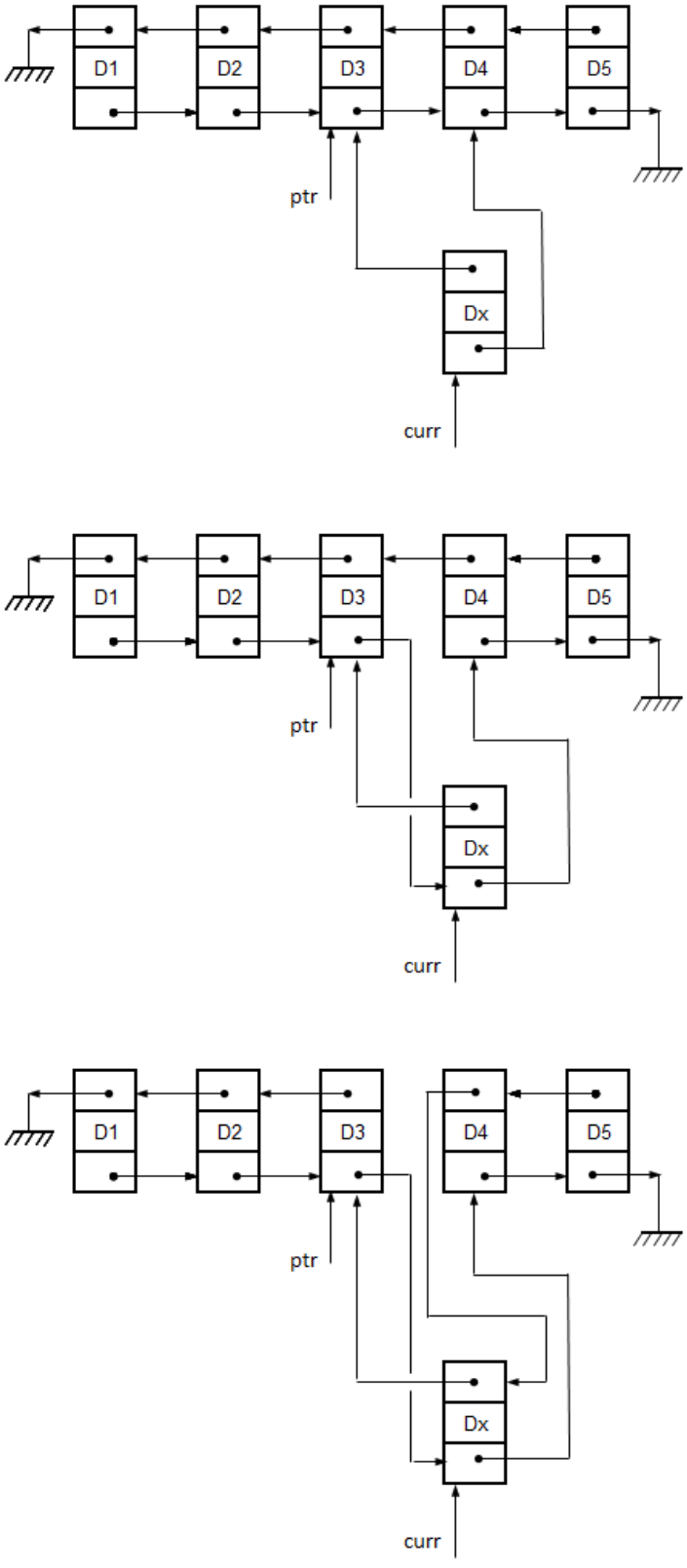
Στο παρακάτω παράδειγμα παρουσιάζεται ενδεικτικά μια συνάρτηση εισαγωγής ενός κόμβου σε συγκεκριμένη θέση μιας διπλά συνδεδεμένης λίστας. Η συνάρτηση δέχεται ως ορίσματα ένα δείκτη ptr σε κάποιο κόμβο της λίστας και ένα ακέραιο, τον pos, ο οποίος ισούται με τον αύξοντα αριθμό του κόμβου από το αριστερό της άκρο, μετά από τον οποίο θα γίνει η εισαγωγή. Οι κόμβοι της λίστας είναι του τύπου dnode, όπως περιγράφεται παραπάνω:

```
void dinsertion(struct dnode *ptr, int pos) {
struct dnode *curr;
int k;

while (ptr -> left != NULL)
     ptr = ptr -> left;
for (k=1; k<pos; k++)
     ptr = ptr -> right;
curr = (struct dnode *) malloc (sizeof (struct dnode));
if (curr != NULL) {
     curr -> right = ptr -> right;
     curr -> left = ptr;
     ptr -> right = curr;
     curr ->right ->left = curr; }}
```

Στο σχήμα 6.3 που ακολουθεί φαίνεται η σειρά των ενεργειών στις οποίες προβαίνουμε για την εισαγωγή κόμβου, σύμφωνα με την παραπάνω συνάρτηση, μετά από τον τρίτο από αριστερά κόμβο της λίστας. Στον κόμβο

αυτόν έχει τοποθετηθεί ο δείκτης ptr με την for που υπάρχει μέσα στην συνάρτηση:




D1
D2
D3
D4
D5
ptr
Dx
curr
D1
D2
D3
D4
D5
ptr
Dx
curr
D1
D2
D3
D4
D5
ptr
Dx
curr

Σχ. 6.3

6.2.3. Ειδικές μορφές λιστών.

α) *Κυκλικές λίστες*. Μπορεί να είναι *απλές κυκλικές* ή *διπλές κυκλικές λίστες*. Είναι γνωστές και ως *δακτύλιοι* (rings). Στις απλές, ο λογικά επόμενος του τελευταίου κόμβου είναι ο πρώτος κόμβος της λίστας, άρα δεν υπάρχει τερματισμός. Ο δείκτης που έδειχνε σε NULL οδηγείται να δείχνει στην κεφαλή της λίστας. Αντίστοιχα, στις διπλές κυκλικές λίστες ο κάθε NULL δείκτης οδηγείται να δείχνει στον κόμβο του άλλου άκρου της λίστας.

β) *Αυτοδιοργανούμενες λίστες* (self organizing lists). Η αναζήτηση στοιχείων σε μια λίστα είναι σπάνια ισοπίθανη. Αν υποθέσουμε ότι γνωρίζουμε την πιθανότητα επίσκεψης κάθε κόμβου μιας απλά συνδεδεμένης λίστας, τότε, διατάσσοντας τους κόμβους της λίστας κατά φθίνουσα σειρά πιθανότητας επίσκεψης, θα είχαμε ταχύτερη αναζήτηση κάποιου στοιχείου. Τίθεται όμως το θέμα του πώς πρέπει να διαταχθούν οι κόμβοι μιας λίστας εάν αυτές οι πιθανότητες δεν είναι εκ των προτέρων γνωστές. Τέτοιες λίστες, οι οποίες αναδιατάσσονται με σκοπό να πλησιάσουν όσο γίνεται την μορφή της φθίνουσας πιθανότητας επίσκεψης λέγονται *αυτοδιοργανούμενες λίστες*. Οι μέθοδοι που εφαρμόζονται για τον σκοπό αυτό είναι οι εξής:

i) *Μετακίνηση στην αρχή* (Move to front method).Σε αυτή την μέθοδο μετακινείται το στοιχείο που αναζητείται στην κεφαλή της λίστας. Εάν το στοιχείο που αναζητείται δεν υπάρχει στην λίστα, τότε δημιουργείται ένας νέος κόμβος, στον οποίο καταχωρείται η αναζητούμενη τιμή και αυτός ο κόμβος τοποθετείται στην κεφαλή της λίστας. Η υλοποίηση είναι απλή, χωρίς επιπλέον απαιτήσεις μνήμης. Έχει το μειονέκτημα ότι πρακτικά μπορεί να αποδώσει σημαντική προτεραιότητα σε μη συχνά εμφανιζόμενα στοιχεία. Προφανώς, εάν η αναζήτηση στοιχείων στην λίστα είναι συχνή, τότε τα στοιχεία με μεγάλη πιθανότητα επίσκεψης συγκεντρώνονται σιγά σιγά στην αρχή της λίστας.

ii) *Μετατόπιση* (Transpose method). Σε αυτή την μέθοδο εναλλάσσεται το στοιχείο που επισημάνθηκε με το προηγούμενό του, εκτός εάν το στοιχείο αυτό είναι το πρώτο της λίστας. Η υλοποίηση είναι επίσης απλή, χωρίς επιπλέον απαιτήσεις μνήμης. Με την μέθοδο αυτή χρειάζονται πολλές προσπελάσεις για την μετακίνηση κάποιου κόμβου στην κεφαλή της λίστας και δεν ενδείκνυται για γρήγορες μεταβολές των πιθανοτήτων επίσκεψης των κόμβων.

iii) *Απαρίθμηση* (Count method). Σε αυτή την μέθοδο μετρώνται οι φορές προσπέλασης κάθε κόμβου. Συνεπώς, για την υλοποίηση της μεθόδου, σε κάθε κόμβο υπάρχει ένας μετρητής, πράγμα που σημαίνει επιπλέον δαπάνη μνήμης. Οι κόμβοι με την μεγαλύτερη συχνότητα προσπέλασης τηρούνται στην κεφαλή ή κοντά στην κεφαλή της λίστας.

6.3. ΔΕΝΤΡΑ.

Είναι μη γραμμικές δομές. Στα δέντρα κάθε στοιχείο έχει ένα μόνο προηγούμενο, αλλά μπορεί να έχει κανένα, ένα ή πολλά επόμενα στοιχεία. Οι κόμβοι ενός δέντρου συνδέονται με ακμές. Συνήθως, η ακμή είναι ένας δείκτης που οδηγεί από τον ένα κόμβο στον άλλο. Ένας από τους κόμβους λέγεται ρίζα(root) και αποτελεί την «αρχή» του δέντρου. Σε κάθε κόμβο του δέντρου καταλήγει μια μόνο ακμή, εκτός από τη ρίζα, στην οποία δεν καταλήγει καμμία. Από την ρίζα μόνο ξεκινούν ακμές. Αντίστοιχα με τις λίστες, πρέπει να γνωρίζουμε ανά πάσα στιγμή την ρίζα, από την οποία μπορούμε να μεταβούμε σε όλους τους κόμβους του δέντρου.

Βαθμός (degree) ενός κόμβου λέγεται ο αριθμός των παιδιών του.

Βαθμός (degree) ενός δέντρου λέγεται ομέγιστος από τους βαθμούς των κόμβων του.

Οι κόμβοι οι οποίοι δεν έχουν παιδιά λέγονται *τερματικοί κόμβοι ή φύλλα* (terminal nodes ή leaves) του δέντρου.

Οι μη τερματικοί κόμβοι λέγονται επίσης και *εσωτερικοί*(internal) ή *κλαδιά* (branches).

Το *επίπεδο (level) κόμβου* ισούται με τον αριθμό των προγόνων του μέχρι τη ρίζα συν 1.

Το *βάθος* (depth) ή *ύψος* (height)*ενός δέντρου*ισούται με το μέγιστο επίπεδο των κόμβων του δέντρου.

Ένα δέντρο έχει πολλά *υποδέντρα* (subtrees) τα οποία προκύπτουν αν θεωρήσουμε κάθε φορά ως ρίζα ένα άλλο κόμβο πλην της πραγματικής αρχικής ρίζας.

Ένα δέντρο λέγεται *πλήρες* (complete) όταν περιέχει τον μέγιστο δυνατό αριθμό κόμβων. Ο μέγιστος αυτός αριθμός εξαρτάται από τον βαθμό και το ύψος του δέντρου. Το πλήθος m των κόμβων στη γενική περίπτωση δέντρου βαθμού p και ύψους k δίνεται από τον τύπο:

$$m = \frac{p^k - 1}{p - 1}$$

Όταν έχει σημασία η διάταξη των κλαδιών ενός δέντρου, αυτό λέγεται *διατεταγμένο* (ordered).

6.3.1. Δυαδικά δέντρα.

Είναι η πιο χρήσιμη μορφή δέντρων. Έχουν βαθμό 2, άρα κανένας κόμβος τους δεν έχει περισσότερα από δύο παιδιά. Η μορφή των κόμβων τους (για ένα δέντρο ακεραίων) περιγράφεται από την δομή:

```
struct  komvos {
    int  data;
    struct komvos *lp;
    structkomvos *rp; };
```

(Σημείωση: Η περιγραφή αυτού του είδους δομών μπορεί να δοθεί και με την χρήση της οδηγίας typedef. Αυτή επιτρέπει τη δημιουργία ονόματος, το οποίο καθορίζει ο χρήστης, για κάποιο τύπο δεδομένων. Η οδηγία typedef μοιάζει με τη #define, αλλά εκτελείται από το μεταγλωττιστή και όχι τον προ-επεξεργαστή και δίνει συμβολικά ονόματα μόνο σε τύπους δεδομένων. Έχει χρησιμοποιηθεί δηλαδή ως εξής στο παράδειγμά μας:

```
typedef struct komvos {
    int  data;
    struct komvos *lp;
    structkomvos *rp; } TREE;
```

Στο εξής δηλαδή οι δομές του τύπου komvos θα έχουν στο πρόγραμμα το όνομα TREE.)

Η διάσχιση ενός δυαδικού δέντρου γίνεται με τρεις ισοδύναμους τρόπους, ανάλογα με το πότε «επισκεπτόμαστε» την ρίζα κάθε υποδέντρου σε σχέση με τα παιδιά της. Αν συμβολίσουμε με Ρ την ρίζα και με Α και Δ τα παιδιά της, τότε διακρίνουμε τους εξής τρόπους διάσχισης με τις αντίστοιχες συναρτήσεις:

α) *Προδιατεταγμένος (Preorder)*με τη σειρά Ρ Α Δ.

```
void preOrder (TREE *q) {
    if (q!=NULL) {
        printf ("%d", q->data);
        preOrder (q->lp);
        preOrder (q->rp); } }
```

β) *Ενδοδιατεταγμένος (Inorder)*με τη σειρά Α Ρ Δ.

```
void inOrder (TREE *q) {
    if (q!=NULL) {
        inOrder (q->lp);
        printf ("%d",q->data);
        inOrder (q->rp); } }
```

γ) *Μεταδιατεταγμένος (Postorder)*με τη σειρά Α Δ Ρ.

```
void postOrder (TREE *q) {
    if (q!=NULL) {
        postOrder (q->lp);
        postOrder (q->rp);
        printf ("%d  ", q->data); } }
```

6.3.2. Δυαδικά δέντρα αναζήτησης.

Αποτελούν ειδική κατηγορία των δυαδικών δέντρων με μια συγκεκριμένη ιδιότητα: Συγκεκριμένα, είναι δέντρα διατεταγμένα, στα οποία *τα δεδομένα κάθε κόμβου έχουν μεγαλύτερη τιμή από τα δεδομένα όλων των απογόνων του που βρίσκονται στα αριστερά του και μικρότερη από την τιμή των δεδομένων όλων των απογόνων του που βρίσκονται δεξιά του*.

Μπορεί κανείς εύκολα να παρατηρήσει, ότι εάν διασχίσουμε ένα δυαδικό δέντρο αναζήτησης (Binary Search Tree, BST) με τον ενδοδιατεταγμένο τρόπο, θα προκύψει αποτέλεσμα ταξινομημένο. Η ταξινόμηση δεν ισχύει για τη διάσχιση του δέντρου με κάποιον από τους άλλους τρόπους.

Τέτοια δέντρα χρησιμοποιούνται όταν θέλουμε γρήγορη προσπέλαση δεδομένων, για τα οποία υπάρχει κατάταξη, τα οποία δηλαδή έχουν ήδη τοποθετηθεί στο δέντρο με βάση κάποιο «κλειδί αναζήτησης». Με αυτό τον όρο εννοούμε ότι από τα πιθανά πολλά «χρήσιμα» δεδομένα του κόμβου, κάποιο (το κλειδί) είναι εκείνο με βάση το οποίο τοποθετούνται ή αναζητούνται στοιχεία μέσα στο δέντρο. Περιγράφουμε και υλοποιούμε στη συνέχεια με C τους αλγόριθμους αναζήτησης στοιχείου, εισαγωγής κόμβου σε ένα BST και διαγραφής κόμβου από ένα BST.

α) Αναζήτηση στοιχείου: Συγκρίνουμε την τιμή της ρίζας του δέντρου με την τιμή που αναζητούμε. Αν η ζητούμενη τιμή είναι μικρότερη από αυτήν της ρίζας, τότε συνεχίζουμε στο αριστερό υποδέντρο. Αν η ζητούμενη τιμή είναι μεγαλύτερη από αυτήν της ρίζας, τότε συνεχίζουμε στο δεξί υποδέντρο. Η αναζήτηση συνεχίζεται μέχρι να βρεθεί ο κόμβος με την τιμή που αναζητούμε ή μέχρι να φθάσουμε σε δείκτη με τιμή NULL, οπότε η αναζητούμενη τιμή δεν υπάρχει στο δέντρο.

Στην παρακάτω συνάρτηση αναζητείται ο κόμβος με τιμή του ακέραιου πεδίου του ίση με num και επιστρέφεται ένας δείκτης στον κόμβο αυτόν ή δείκτης ίσος με NULL, εάν το num δεν υπάρχει στο δέντρο:

```
TREE *search (TREE *root, intnum) {
    TREE *ptr;

    ptr = root;
    while (ptr->data != snum) {
        if (ptr->data > snum) {
            ptr = ptr->lp;
            if (ptr == NULL)
                break; }
        if (ptr->data < snum) {
            ptr = ptr->rp;
```

```
                if (ptr == NULL)
                    break; } }
        return ptr; }
```

β) Εισαγωγή κόμβου: Εφαρμόζεται η διαδικασία αναζήτησης, αναζητώντας ένα κόμβο με τιμή ίση με αυτή του κόμβου που πρόκειται να εισαχθεί. Τέτοια τιμή δεν υπάρχει και καταλήγουμε σε δείκτη NULL. Δεσμεύεται μνήμη για την τοποθέτηση της νέας τιμής και δημιουργείται ένας νέος κόμβος. Ο μηδενικός δείκτης στον οποίο καταλήξαμε πιο πάνω τοποθετείται έτσι ώστε να δείχνει στο νέο κόμβο.

```
void insert (TREE *root, int ak) {
    TREE *ptr, *dkt;

    ptr = root;
    dkt = (TREE *) malloc (sizeof(int));
    dkt -> data = ak;
    dkt -> rp = NULL;
    dkt -> lp = NULL;
    while (ptr -> data != ak) {
        if (ak < ptr -> data) {
            if (ptr -> lp != NULL)
                ptr = ptr -> lp;
            else
                ptr -> lp = dkt; }
        if (ak > ptr -> data) {
            if (ptr -> rp != NULL)
                ptr = ptr -> rp;
            else
                ptr -> rp = dkt; }}}
```

γ) Διαγραφή κόμβου: Αν ο υπό διαγραφή κόμβος είναι φύλλο του δέντρου, τότε τον διαγράφουμε, μηδενίζοντας τον αντίστοιχο δείκτη ο οποίος ξεκινά από τον πατέρα του. Αν ο υπό διαγραφή κόμβος έχει ένα μόνο παιδί, τότε αντικαθίσταται από το παιδί του. Αν ο υπό διαγραφή κόμβος έχει δύο παιδιά, τότε αντικαθίσταται είτε από τον πιο δεξιό κόμβο του αριστερού υποδέντρου, είτε από τον πιο αριστερό κόμβο του δεξιού υποδέντρου.

```
void del (TREE *root, int ak) {
    TREE *ptr, *dkt, *curr, *temp;
    ptr = root;
    while (ak != ptr -> data) {
        dkt = ptr;                          // O dkt  στον γονέα του ptr
        if (ak < ptr -> data)
            ptr = ptr -> lp;
        else
            ptr = ptr -> rp; }
```

```
if (ptr -> lp == NULL && ptr -> rp == NULL) {// Ο ptrείναιφύλλο
    if (dkt -> lp == ptr)
        dkt -> lp = NULL;
    if (dkt -> rp == ptr)
        dkt -> rp = NULL;
    free (ptr); }
else
    if (ptr -> lp == NULL && ptr -> rp != NULL) {    // Ο ptrέχει
        if (dkt ->lp == ptr)                          // μόνοδεξίπαιδί
            dkt -> lp = ptr -> rp;
        if (dkt -> rp == ptr)
            dkt -> rp = ptr -> rp;
        ptr -> rp = NULL;
        free (ptr); }
    else
        if (ptr -> rp == NULL && ptr -> lp != NULL) {    // Optrέχει
            if (dkt ->lp == ptr)// μόνοαριστερόπαιδί
                dkt -> lp = ptr -> lp;
            if (dkt -> rp == ptr)
                dkt -> rp = ptr -> lp;
            ptr -> lp = NULL;
            free (ptr); }

        else// Optrέχει δύο παιδιά
            if (ptr -> lp != NULL && ptr -> rp != NULL) {
                curr = ptr;
                temp = curr;
                curr = curr -> rp;
                while (curr -> lp != NULL) {
                    temp = curr;
                    curr = curr -> lp; }
                ptr -> data = curr -> data;
                temp -> lp = NULL;
                free (curr); }}
```

6.3.3. Νηματικά δέντρα.

Σε ένα δυαδικό δέντρο (γενικά, δεν μιλάμε μόνο για δέντρο αναζήτησης) υπάρχουν δείκτες, από τους οποίους κάποιοι έχουν τιμή NULL. Οι δείκτες αυτοί είναι δυνατό να αξιοποιηθούν με τέτοιο τρόπο, ώστε να μην έχουν τιμή NULL, αλλά να δείχνουν σε συγκεκριμένους κόμβους μέσα στο δέντρο.

Για τον σκοπό αυτό, ο δεξιός δείκτης κάθε κόμβου, όταν έχει τιμή NULL, μετατρέπεται έτσι ώστε να δείχνει στον επόμενο κόμβο του δέντρου, λαμβάνοντας υπ' όψη τον ενδοδιατεταγμένο τρόπο διάσχισης. Ένα τέτοιο δέντρο λέγεται *δεξιό ενδονηματικό*.Ως κύρια πλεονεκτήματα του αναφέρουμε πρώτα-πρώτα την εύκολη διάσχιση με ενδοδιατεταγμένο τρόπο. Χωρίς την χρήση νημάτων, η διάσχιση πρέπει να υλοποιηθεί είτε αναδρομικά είτε με τη

χρήση στοίβας. Μπορούμε επίσης να βρεθούμε στον γονέα κάποιου κόμβου χωρίς να χρειάζεται η ύπαρξη ξεχωριστού δείκτη στον γονέα

Ένα πρόβλημα που εμφανίζεται σε τέτοιου είδους δέντρα είναι κατά πόσο ένας δείκτης είναι κανονικός δείκτης ή δείκτης-νήμα. Το πρόβλημα μπορεί να αντιμετωπιστεί με την εισαγωγή ενός επιπλέον πεδίου. Αν αυτό έχει την τιμή 1, τότε ο δεξιός δείκτης είναι νήμα, αλλιώς είναι κανονικός δείκτης πρς απόγονο.

Το *αριστερό ενδονηματικό δέντρο* υλοποιείται με παρόμοιο τρόπο, αλλά εδώ ο αριστερός δείκτης κάθε κόμβου, όταν έχει τιμή NULL, μετατρέπεται έτσι ώστε να δείχνει στον προηγούμενο κόμβο του δέντρου, λαμβάνοντας υπ' όψη τον ενδοδιατεταγμένο τρόπο διάσχισης.

Το δέντρο μπορεί να υλοποιηθεί και με τους δύο τρόπους ταυτόχρονα, οπότε λέγεται απλά *ενδονηματικό*. Με ανάλογο τρόπο προκύπτουν τα *προνηματικά δέντρα* και τα *μετανηματικά δέντρα* (δεξιά, αριστερά ή απλά), αν ληφθεί υπ' όψη η προδιατεταγμένη ή η μεταδιατεταγμένη διάσχιση αντίστοιχα.

Στο παράδειγμα που ακολουθεί, διαθέτουμε ένα αριστερό ενδονηματικό δέντρο ακεραίων. Η δομή που περιγράφει τους κόμβους του δέντρου είναι:

```
typedef struct komvos {
    int ak;
    struct  komvos *ap;
    struct  komvos *dp;
    unsigned int  thr: 1; } TREE;
```

Η παρακάτω συνάρτησηarnima( ) γράφει στην οθόνη τους ακεραίους των κόμβων, στους οποίους οδηγούμαστε μέσω νημάτων αν ξεκινήσουμε από τον άκρο δεξιό κόμβο του δέντρου μέχρι την ρίζα του, ακολουθώντας πάντα δείκτες προς αριστερά παιδιά. Η συνάρτηση επιστρέφει ένα δείκτη στον πιο αριστερόκόμβο του δέντρου.

```
TREE *ar_nima (TREE *root) {
    TREE *ptr, *temp;

    ptr = root;
    while (ptr ->dp != NULL)
        ptr = ptr ->dp;
    temp = ptr;
    while (ptr != root) {
        if (ptr -> thr == 1) {
                ptr = ptr ->ap;
                printf ("%d", ptr -> ak); }
        else
                ptr = ptr -> ap; }
    return temp;
```

Οι επόμενες δύο συναρτήσεις μας δίνουν τη δυνατότητα διάσχισης με ενδοδιατεταγμένο τρόπο ενός ενδονηματικού δέντρου χωρίς τη χρήση αναδρομής ή στοίβας όπως είπαμε πιο πάνω.

Η συνάρτηση **full_left( )** οδηγεί το όρισμά της στον πιο αριστερό κόμβο του υποδέντρου, το οποίο έχει ως ρίζα το όρισμά της. Στη συνέχεια, η συνάρτηση **traverse( )** διασχίζει με ενδοδιατεταγμένο τρόπο το δέντρο, το οποίο έχει ως ρίζα το όρισμά της:

```
TREE *full_left (TREE *node) {
    if (node == NULL)
        return NULL;
    while (node -> ap != NULL)
        node = node -> ap;
    return node;}

void traverse (TREE *root) {
    TREE *cur = full_left (root);
    while (cur != NULL) {
        printf ("%d", cur -> ak);
        if (cur -> thr)
            cur = cur -> dp;
        else
            cur = full_left (cur->dp); }}
```

6.3.4. Ψηφιακά δέντρα.

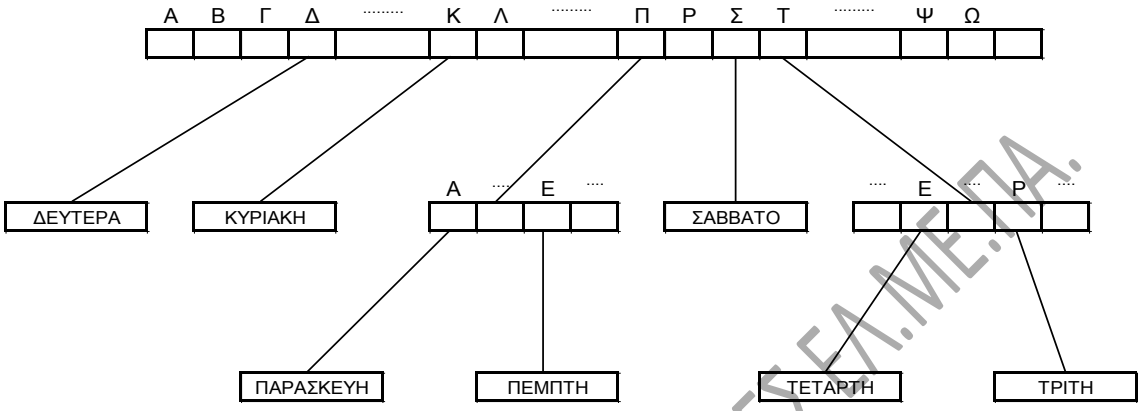
Τα *ψηφιακά δέντρα* έχουν την ιδιαιτερότητα ότι οι κόμβοι τους έχουν τιμές που ανήκουν σε ένα συγκεκριμένο σύνολο τιμών, π.χ. τα γράμματα της αλφαβήτου, οι αριθμοί 0 έως 9 κλπ. Η πληροφορία η οποία μεταφέρεται στα δέντρα αυτά σχηματίζεται από την διαδρομή που ακολουθούμε αν διατρέξουμε το δέντρο ξεκινώντας από την ρίζα μέχρι κάποιο φύλλο του. Ένα πολύ σημαντικό πλεονέκτημα των δέντρων αυτών είναι ότι μπορούν να αποθηκεύσουν πληροφορίες μεταβλητού μήκους.

6.3.5. Δομή trie.

Η δομή *trie* είναι μια υλοποίηση των ψηφιακών δέντρων για αλφαβητικά κυρίως δεδομένα και χρησιμοποιείται πάρα πολύ σε λεξικά, ελεγκτές ορθογραφίας κλπ. Η λέξη προέρχεται από την λέξη re-trie-val που σημαίνει ανάκτηση, προφέρεται όμως «τράι». Στις δομές αυτές, τα φύλλα του δέντρου αποτελούν τους κόμβους πληροφορίας, ενώ οι εσωτερικοί κόμβοι είναι απλώς κλαδιά του δέντρου. Στους κόμβους των δέντρων trie υπάρχουν μόνο δείκτες προς

συγκεκριμένες θέσεις και η πληροφορία εξάγεται έμμεσα από την θέση στην οποία δείχνει ο δείκτης.

Στο παρακάτω σχήμα παρουσιάζεται μια δομή trie, στην οποία οι κόμβοι πληροφορίας είναι τα ονόματα των ημερών της εβδομάδας:



Σχ. 6.4

Επειδή η καταχωρούμενη πληροφορία είναι αλφαβητική, κάθε κόμβος κλαδί περιέχει ένα πίνακα 25 δεικτών, όσα δηλαδή τα γράμματα του αλφαβήτου και το κενό, ενώ οι δείκτες που δεν εμφανίζονται στο σχήμα αυτό (π.χ. οι δείκτες από τις θέσεις Α, Β, Γ κλπ της ρίζας του δέντρου) έχουν τιμή NULL. Παρατηρείστε πώς για παράδειγμα από τον δείκτη της θέσης Π του πρώτου κόμβου (ρίζας) οδηγούμαστε στον δείκτη της θέσης Ε του δεύτερου και από εκεί -αναγκαστικά και μόνο- στην λέξη ΠΕΜΠΤΗ.

Είναι φανερό ότι αν οι καταχωρημένες «τιμές» (οι μέρες της εβδομάδας εν προκειμένω) είναι λίγες και αραιές, τότε υπάρχει μεγάλη σπατάλη χώρου και έτσι η δομή αυτή είναι μη συμφέρουσα. Φυσικά, αν κάποιες τιμές είναι δεδομένο ότι δεν υπάρχουν, τότε, ανάλογα με τον αλγόριθμο υλοποίησης του δέντρου, είναι πιθανόν να μην υπάρχουν και οι αντίστοιχοι κόμβοι.

Πλεονέκτημα του δέντρου αυτού αποτελεί το γεγονός ότι μπορούμε να αναζητήσουμε και να εισαγάγουμε συμβολοσειρές σε χρόνο O(m), όπου m το μήκος της συμβολοσειράς. Ακόμα και στο BST η πολυπλοκότητα είναι ης τάξης του O(n).

Παρατηρείστε ενδεικτικά κώδικα κάποιων συναρτήσεων για χειρισμό του δέντρου:

Η περιγραφή της δομής για ένα κόμβο ενός δέντρου trie μπορεί να είναι η παρακάτω:

```
typedef  struct  trie {
   int  leaf;
   char name[20];
   struct  trie *pin[26]; }TRIE;
```

Το πεδίο leaf θα τίθεται 1 ή 0 ανάλογα με το εάν ο κόμβος είναι φύλλο του δέντρου ή εσωτερικός κόμβος. Συνεπώς, ακόμη οικονομικότερα, η δομή αυτή θα μπορούσε να περιγραφεί ως εξής:

```
typedef  struct  trie {
   unsigned  leaf : 1;
   char  name[20];
   struct  trie  *pin[26]; }TRIE;
```

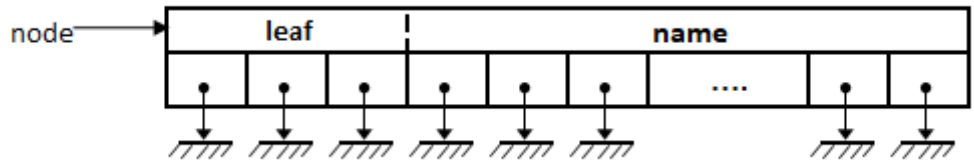
Στο πεδίο name θα φυλάσσεται η τιμή που υπάρχει στο φύλλο, αλλιώς θα περιέχει μια κενή συμβολοσειρά.

Το πεδίο pin θα είναι ένας πίνακας δεικτών σε δομές του είδους TRIE. Καθένας από αυτούς τους δείκτες θα δείχνει σε ένα από τους επόμενους εσωτερικούς κόμβους του δέντρου.

Η συνάρτηση που ακολουθεί δημιουργεί ένα κόμβο του δέντρου:

```
TRIE* new_node ( ) {
     TRIE* node;
     int k;

     node = (TRIE *) malloc (sizeof (TRIE));
     node -> leaf = 0;
     for (k = 0; k< SIZE; k++)
          node -> pin[k] = NULL;
     return node; }
```

δηλαδή ένα κόμβο, ο οποίος σχηματικά παριστάνεται ως:



Σχ. 6.5

Η παρακάτω συνάρτηση εισάγει μια συμβολοσειρά αποτελούμενη από μικρούς λατινικούς χαρακτήρες στο δέντρο trie:

```
void insert (TRIE *root, char* str) {
    TRIE * curr = root;

    while (*str) {
        if (curr -> pin[*str - 'a'] == NULL)
            curr->pin[*str - 'a'] = new_node( );
        curr = curr -> pin[*str - 'a'];
        str++; }

    curr -> leaf = 1;
    strcpy (curr -> name, str); }
```

6.4. ΓΡΑΦΟΙ.

Ο *γράφος* (graph) είναι μια μη γραμμική δομή δεδομένων. Χαρακτηρίζεται από δυο σύνολα, τα V και E. Το V είναι ένα πεπερασμένο σύνολο, το V={v1, v2,…, vn}, με n>0, τα στοιχεία του οποίου είναι οι κορυφές (κόμβοι, vertices) του γράφου, οι οποίες χρησιμοποιούνται για την παράσταση των δεδομένων. Αντίστοιχα, το E={(x,y), όπου x, y ε V} είναι το σύνολο των ακμών (edges) του γράφου, οι οποίες παριστάνουν τις σχέσεις μεταξύ των δεδομένων. Είναι προφανές λοιπόν ότι μια ακμή «περιγράφεται» από το ζεύγος των κορυφών τις οποίες συνδέει. Ο γράφος μπορεί να συμβολιστεί ως G(V,E) και ορίζεται πλήρως από τα σύνολα V και E. Αυτό σημαίνει ότι δυο γράφοι μπορεί να είναι εντελώς ταυτόσημοι, παρά το ότι η μορφή τους, η θέση των κορυφών τους και το μήκος των ακμών στη γεωμετρική τους παράσταση μπορεί να διαφέρουν.

Ένας γράφος λέγεται *ζυγισμένος*, όταν κάθε ακμή του χαρακτηρίζεται από ένα αριθμό (βάρος).

Ένας γράφος λέγεται *μη κατευθυνόμενος*, αν οι ακμές του δεν είναι προσανατολισμένες προς κάποια κατεύθυνση, αν δηλαδή τα ζεύγη (v_1, v_2) και (v_2, v_1) παριστάνουν την ίδια ακμή. Αντίθετα, αν τα διάφορα ζεύγη (v_i, v_j) έχουν διάταξη, τότε ο γράφος λέγεται *κατευθυνόμενος*.

Ο *μέγιστος αριθμός ακμών* για ένα μη κατευθυνόμενο γράφο με n κορυφές είναι n*(n-1)/2, ενώ ο αντίστοιχος αριθμός για ένα κατευθυνόμενο γράφο είναι n*(n-1).

Αν (v_1, v_2) είναι μια ακμή ενός μη κατευθυνόμενου γράφου, τότε οι κορυφές v_1 και v_2 λέγονται *γειτονικές* και η ακμή (v_1, v_2) λέγεται προσκείμενη στις κορυφές v_1 και v_2. Αν μια ακμή ενός κατευθυνόμενου γράφου είναι η $<v_1, v_2>$, τότε η v_1 λέγεται *γειτονική προς* την v_2 ενώ η v_2 λέγεται *γειτονική από* την v_1.

Βαθμός κορυφής ενός μη κατευθυνόμενου γράφου λέγεται ο αριθμός των ακμών που είναι προσκείμενες της κορυφής. Σε ένα κατευθυνόμενο γράφο αναφέρονται ο *βαθμός εισόδου* και ο *βαθμός εξόδου* για τον αριθμό των ακμών που καταλήγουν σε μια κορυφή και τον αριθμό των ακμών που ξεκινούν αντίστοιχα από μια κορυφή.

Στη συνέχεια δίνονται δύο τρόποι υλοποίησης γράφων.

6.4.1. Πίνακες γειτονικών κορυφών.

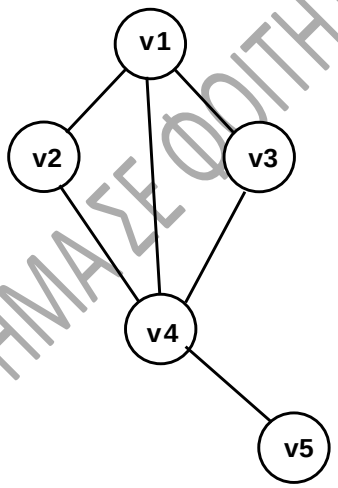
Για ένα μη κατευθυνόμενο γράφο G με n κορυφές ο πίνακας γειτονικών κορυφών είναι ένας n x n τετραγωνικός, συμμετρικός πίνακας, έστω ο pin, για τον οποίο ισχύει:

$pin_{j\,k}$ = 1, αν η κορυφή v_j είναι γειτονική με την κορυφή v_kκαι

$pin_{j\,k}$ = 0, αν η κορυφή v_j δεν είναι γειτονική με την κορυφή v_k

Από τον γειτονικό πίνακα μπορεί να προκύψει ο βαθμός κάποιας κορυφής ενός μη κατευθυνόμενου γράφου, αθροίζοντας τα περιεχόμενα της αντίστοιχης σειράς του πίνακα.

Στο σχ. 6.6 παρουσιάζονται ένας μη κατευθυνόμενοςγράφος και ο πίνακας γειτονικών κορυφών του:



	v1	v2	v3	v4	v5
v1	0	1	1	1	0
v2	1	0	0	1	0
v3	1	0	0	1	0
v4	1	1	1	0	1
v5	0	0	0	1	0

Σχ. 6.6

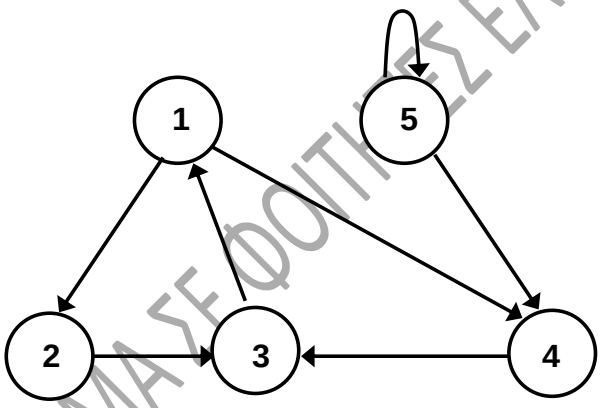
Αν ο γράφος είναι κατευθυνόμενος, τότε για τον πίνακα γειτονικών κορυφών ισχύει ότι:

$pin_{j\,k}$ = 1, αν η κορυφή v_kείναι γειτονική από την κορυφή v_jκαι

$pin_{j\,k}$ = 0, αν η κορυφή v_kδεν είναι γειτονική από την κορυφή v_j

Σε ένα κατευθυνόμενο γράφο, ο βαθμός εισόδου της κορυφής k δίνεται από το άθροισμα των τιμών της στήλης k του πίνακα γειτονικών κορυφών, ενώ ο βαθμός εξόδου της κορυφής k δίνεται από το άθροισμα των τιμών της γραμμής k του πίνακα.

Στο σχ. 6.7 παρουσιάζονται ένας κατευθυνόμενοςγράφος και ο πίνακας γειτονικών κορυφών του:



	v1	**v2**	**v3**	**v4**	**v5**
v1	0	1	0	1	0
v2	0	0	1	0	0
v3	1	0	0	0	0
v4	0	0	1	0	0
v5	0	0	0	1	1

Πίνακας μονοπατιών μήκους 2:

	v1	**v2**	**v3**	**v4**	**v5**
v1	0	0	2	0	0
v2	1	0	0	0	0
v3	0	1	0	1	0
v4	1	0	0	0	0
v5	0	0	1	1	1

Πίνακας μονοπατιών μήκους 3:

	v1	**v2**	**v3**	**v4**	**v5**
v1	2	0	0	0	0
v2	0	1	0	1	0
v3	0	0	2	0	0
v4	0	1	0	1	0
v5	1	0	1	1	1

Σχ. 6.7

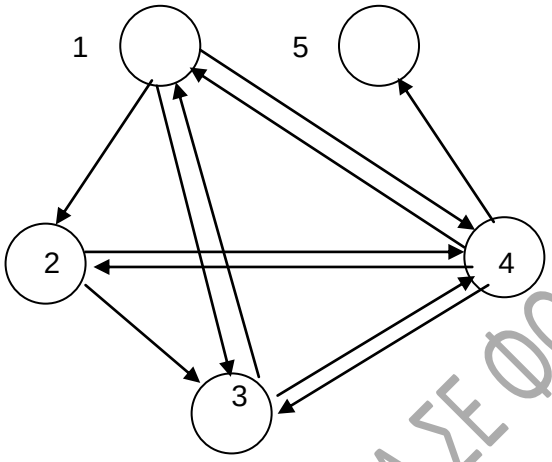
Εάν πολλαπλασιάσουμε τον πίνακα γειτονικών κορυφών με τον εαυτό του και κάνοντας χρήση ιδιοτήτων των πινάκων και στοιχείων Γραμμικής Άλγεβρας, προκύπτει ένας νέος πίνακας ίδιων προφανώς διαστάσεων. Ο πίνακας αυτός ονομάζεται *πίνακας μονοπατιών μήκους 2* (βλ. σχ. 6.7). Τα στοιχεία του είναι διάφορα του μηδενός μόνο εάν υπάρχει μονοπάτι μήκους 2, το οποίο να συνδέει τις δύο κορυφές που συσχετίζει το συγκεκριμένο στοιχείο. Ομοίως, πολλαπλασιάζοντας τον πίνακα μονοπατιού μήκους 2 με τον αρχικό πίνακα, καταλήγουμε στον *πίνακα μονοπατιών μήκους 3* (βλ. σχ. 6.7), δηλαδή ένα πίνακα που περιγράφει εάν υπάρχει μονοπάτι μήκους 3 μεταξύ δύο κορυφών κ.ο.κ.

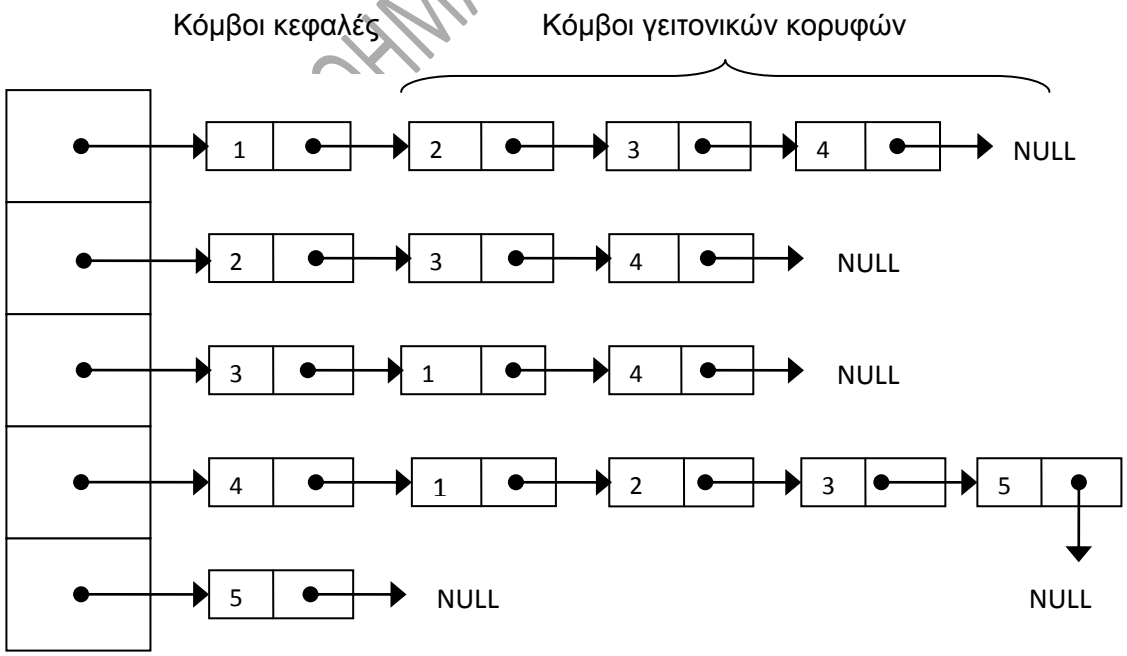
Από τα παραπάνω καταλαβαίνουμε ότι δεν μπορούμε να διαγράψουμε μια κορυφή ή να εισαγάγουμε μια νέα στον γράφο, αφού οι πίνακες αποτελούν «στατικές» δομές. Πρέπει επίσης να σημειωθεί ότι αν ο γράφος είναι αραιός, η παράσταση με πίνακα γειτονικών κορυφών οδηγεί σε μεγάλη σπατάλη μνήμης. Στον κώδικα που ακολουθεί δημιουργείται ένας γράφος 100 το πολύ κορυφών Στο πρόγραμμα ζητείται το πλήθος των κορυφών που θα έχει ο γράφος, το πλήθος των ακμών και το βάρος κάθε ακμής. Ο γράφος παριστάνεται με τον πίνακα γειτονικών κορυφών του και παριστάνεται στην οθόνη μέσω του πίνακα αυτού:

```
#define N 100

void create_graph (int [ ][N], int, int);
void display_graph (int [ ][N], int);

int main( ) {
     int adj[N][N], v, e;

     printf ("Enter vertices and Edges:\n");
     scanf ("%d%d", &v, &e);
     if (e > v*(v-1)/2)
          printf ("Too many edges\n");
     else {
           create_graph (adj, v, e);
           display_graph (adj, v); }

     return 0; }


void create_graph (int adj[ ][N], int v, int e) {
     int i, j;
     int beg, end, val;

     for (i = 0; i < v; i++)                          // Reset graph
           for (j = 0; j < v; j++)
                adj[i][j] = 0;
     for (i = 0; i < e; ++i) {                        // Input graph
          printf ("Enter %d edge (Begin  End  Value):\n", i+1);
          scanf ("%d%d%d", &beg, &end, &val);
          adj[beg-1][end-1] = adj[end-1][beg-1] = val;} }


void display_graph (int adj[ ][N], int v) {
     int i, j;

     printf ("\nGraph:\n");
     for (i = 0; i < v; i++) {
           for (j = 0; j < v; j++)
                printf ("%5d", adj[i][j]);
           printf("\n"); }
     printf("\n"); }
```

6.4.2. Λίστες γειτονικών κορυφών.

Στην παράσταση ενός γράφου με λίστες γειτονικών κορυφών, σε κάθε κορυφή του γράφου αντιστοιχούμε μια απλά συνδεδεμένη λίστα, η οποία περιέχει όλες τις γειτονικές κορυφές της αρχικής. Ο γράφος υλοποιείται με την χρήση ενός πίνακα δεικτών, κάθε στοιχείο του οποίου είναι ένας δείκτης προς ένα κόμβο μιας λίστας. Η κεφαλή της λίστας περιέχει τον αριθμό της κορυφής του γράφου και ένα δείκτη προς τον επόμενο κόμβο, ο οποίος με την σειρά του περιέχει τον αριθμό μιας κορυφής γειτονικής προς την αρχική κοκ.





Πίνακας δεικτών

Σχ. 6.8

Στα σχήμα 6.8 απεικονίζεται ένας προσανατολισμένος γράφος και η παράστασή του με λίστες γειτονικών κορυφών.

Το πρόγραμμα που ακολουθεί δημιουργεί την λίστα γειτονικών κορυφών για ένα γράφο, το πλήθος των κόμβων του οποίου ζητείται και δίνεται από τον χρήστη:

```
typedef struct edge {
      int beg;
      int end; } EDGE;

typedef struct komvos{
      int ak;
      struct komvos *next; } KOMVOS;

int main ( ) {
      EDGE item;
      KOMVOS **ptr, *curr;
      int n, k, j;

      printf ("Give number of nodes ");
      scanf ("%d",&n);
      ptr = (KOMVOS **) malloc (n * sizeof (KOMVOS *)); /*Δημιουργία πίνακα
                                                          δεικτών */
      for (k=0; k<n; k++){                          /* Δημιουργία πρώτων κόμβων */
            *(ptr+k) = (KOMVOS *) malloc ( sizeof (KOMVOS));
            (*(ptr+k)) ->ak = k;                    /*Τιμές στους πρώτους κόμβους */
            (*(ptr+k)) ->next = NULL;}

      printf ("Source ");                           /* Αρχή ακμής και λίστας */
      scanf ("%d", &item.beg);
      while (item.beg >= 0) {
            printf ("Destination ");
            scanf ("%d", &item.end);                /* Τέλοςακμής */
            curr = *(ptr + item.beg);               /* Στην θέση του πίνακα δεικτών */
                                                    /* όση η τιμή του κόμβου αρχής */
            while (curr ->next != NULL)             /* Νέος κόμβος στο τέλος της λίστας */
                  curr = curr ->next;
            curr -> next = (KOMVOS *) malloc (sizeof(KOMVOS));
            curr ->next ->ak = item.end;
            curr -> next -> next = NULL;
            printf ("Source ");                     /* Αρχή νέας ακμής και λίστας */
            scanf ("%d", &item.beg); }

      j = 0;
      while (j<n) {                                 /* Εμφάνιση κορυφών γράφου */
            curr = *(ptr + j);
            while (curr != NULL) {
                  printf("%5d", curr -> ak);
                  curr = curr -> next; }

            printf("\n");
            j++; }

      return 0; }
```

6.5. ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΣ.

Ο κατακερματισμός είναι μια λειτουργία, η οποία χρησιμοποιεί μια ειδική συνάρτηση («συνάρτηση κατακερματισμού»), έτσι ώστε να αντιστοιχίζει μια τιμή (θα την λέμε «κλειδί» στη συνέχεια) σε μια θέση ενός πίνακα, κάνοντας με τον τρόπο αυτό ταχύτερη την αναζήτηση στοιχείων. Η αποδοτικότητα της αντιστοίχισης εξαρτάται από την συνάρτηση που χρησιμοποιείται.

Ένας πίνακας κατακερματισμού αποτελείται από ένα σύνολο από θέσεις, σε κάθε μια από τις οποίες αποθηκεύεται ένα κλειδί, αφού πρώτα περάσει από την συνάρτηση κατακερματισμού. Αν το κλειδί είναι αριθμητικό, η συνάρτηση είναι συνήθως μια απλή παράσταση. Αν το κλειδί είναι αλφαριθμητικό, τότε με κάποιο τρόπο πρέπει να μετατραπεί σε αριθμητικό (π.χ. με την χρήση του κώδικα ASCII).

6.5.1. Συναρτήσεις κατακερματισμού.

Αναφέρουμε κάποια παραδείγματα συχνά χρησιμοποιούμενων συναρτήσεων κατακερματισμού.

α) Διαίρεση με πρώτο αριθμό. Η θέση του πίνακα όπου θα αποθηκευτεί το κλειδί προκύπτει από το υπόλοιπο της διαίρεσης του κλειδιού (modulo) με το μέγεθος του πίνακα. Προφανώς μπορεί να ληφθεί το υπόλοιπο της διαίρεσης του κλειδιού με άλλο ακέραιο, αρκεί να είναι μικρότερος από το μέγεθος του πίνακα. Είναι συνηθισμένο (και όπως προκύπτει από παρατηρήσεις, αρκετά αποδοτικό), ο διαιρέτης να είναι ο μεγαλύτερος πρώτος αριθμός, ο οποίος είναι μικρότερος από το μέγεθος του πίνακα.

Έστω για παράδειγμα ένα σύνολο τιμών S = {5, 14, 25, 44, 68, 99, 150, 373, 401, 509}, τις οποίες θα αποθηκεύσουμε σε ένα πίνακα κατακερματισμού Η, 7 θέσεων. Εφαρμόζουμε σε κάθε κλειδί x την συνάρτηση κατακερματισμού x % 7. Έτσι προκύπτει ο παρακάτω πίνακας κατακερματισμού:

0	14		
1	99		
2	44	373	401
3	150		
4	25		
5	5	68	509
6			

Η συνάρτηση κατακερματισμού θα μπορούσε να είναι η πιο κάτω συνάρτηση, η οποία θα δέχεται ως είσοδο μια συμβολοσειρά και θα επιστρέφει ένα ακέραιο, ο οποίος θα ισούται με το μήκος της συμβολοσειράς modulo το μέγεθος του πίνακα κατακερματισμού:

```
int hash (char *arr, int size) {
   int ak;

   ak = strlen (arr);
   return  ak % size; }
```

Με την ίδια λογική γίνεται και η αναζήτηση στοιχείων. Αν για παράδειγμα αναζητούμε τον αριθμό 68, πρέπει να ψάξουμε στην θέση 5, το στοιχείο δηλαδή Η[5] (το 68 % 7).

β) Μετατροπή ρίζας (radix conversion). Εάν το κλειδί δεν είναι αριθμός του δεκαδικού αριθμητικού συστήματος, τότε μετατρέπεται σε δεκαδικό αριθμό και το τελευταίο ψηφίο δηλώνει την θέση αποθήκευσης στον πίνακα. Μπορεί επίσης η θέση αποθήκευσης να προκύψει από την πράξη modulo του κλειδιού με το μέγεθος του πίνακα.

Δίνουμε ένα σχετικά ρεαλιστικό παράδειγμα, θεωρώντας ότι τα κλειδιά ανήκουν στο δεκαεξαδικό σύστημα και η αποθήκευση γίνεται σε ένα πίνακα 100 θέσεων. Κάθε κλειδί αποθηκεύεται στην θέση που καθορίζουν τα δύο τελευταία ψηφία του αριθμού. Παρακάτω εμφανίζονται τα κλειδιά και οι θέσεις αποθήκευσης:

Κλειδί	Δεκαδικός	Θέση
13F8	6648	48
273A	10042	42
5414	21524	24
3245	12869	69
105B	4187	87
A03C	41020	20
400F	16399	99
06FD	1789	89

γ) Αναδίπλωση (folding). Το κλειδί χωρίζεται σε δύο μέρη, τα οποία προστίθενται μεταξύ τους. (Σε μια παραλλαγή της μεθόδου προστίθεται το πρώτο τμήμα με το δεύτερο αφού αναστραφούν τα ψηφία του). Η θέση αποθήκευσης μπορεί για παράδειγμα να προκύψει από τα δύο τελευταία ψηφία

του αριθμού, όπως επίσης μπορεί να προκύψει από την πράξη modulo του κλειδιού με το μέγεθος του πίνακα.

Στο παράδειγμα που ακολουθεί εμφανίζονται τα κλειδιά και οι θέσεις αποθήκευσης στον πίνακα, με αναδίπλωση μετά το τρίτο στοιχείο του κλειδιού:

Κλειδί	Θέση
56349	12
10042	42
21524	39
12869	97
23566	01
41020	30
16399	62
36445	09

δ) Τετράγωνο του μέσου (mid square). Λαμβάνουμε τα μεσαία ψηφία του κλειδιού, τον αριθμό που προκύπτει τον υψώνουμε στο τετράγωνο και παίρνουμε τα μεσαία ψηφία αυτού του αριθμού.

Στο παράδειγμα που ακολουθεί λαμβάνουμε τα τρία μεσαία ψηφία του κλειδιού και τα δύο μεσαία ψηφία του τετραγώνου:

Κλειδί	Τετράγωνο	Θέση
56349	401956	19
10042	000016	00
21524	023104	31
12869	081796	17
23566	126736	67
41020	010404	04
16399	408321	83
36445	414736	47

Δύο κύρια ερωτήματα που τίθενται είναι: α) Ποια είναι η καλύτερη επιλογή για την συνάρτηση κατακερματισμού και β) Τι θα γίνει εάν πολλοί αριθμοί αποθηκεύονται στην ίδια θέση (δείτε ας πούμε στο παράδειγμα της πιο πάνω παραγράφου (α) τους

αριθμούς 5, 68 και 509), κάτι που είναι πολύ πιθανό, οπότε μιλούμε για *συγκρούσεις* (collisions).

Μια καλή συνάρτηση κατακερματισμού θα πρέπει:

- Να χρησιμοποιεί όλο τον πίνακα κατακερματισμού
- Να κατανέμει ομοιόμορφα τις τιμές στον πίνακα και
- Να είναι απλή ως προς τον υπολογισμό της.

Προφανώς μια συνάρτηση κατακερματισμού μπορεί να μην έχει ως είσοδο ένα αριθμό. Στο παρακάτω παράδειγμα, η συνάρτηση κατακερματισμού δέχεται ως είσοδο μια συμβολοσειρά:

```
int hash(char *s) {
    int sum=0;

    for(*s; *s; s++)
        sum ^= *s;
    return (sum& 127);}
```

Στο παράδειγμα αυτό, για παράδειγμα η συμβολοσειρά εισόδου BLUE θα δώσει αποτέλεσμα 30, ενώ η GREEN θα δώσει αποτέλεσμα 91.

6.5.2. Αντιμετώπιση συγκρούσεων με αλυσίδες.

Όπως αναφέρθηκε, περισσότερες από μια τιμές μπορούν να έχουν το ίδιο αποτέλεσμα εάν εφαρμοστεί σε αυτές η συνάρτηση κατακερματισμού. Μπορούμε να θεωρήσουμε ότι κάθε θέση του πίνακα κατακερματισμού δείχνει σε μια απλά συνδεδεμένη λίστα. Για να βρούμε ένα κλειδί, το k, θα ψάξουμε στην λίστα, η οποία δείχνεται από την θέση H[k]. ΟΗ δηλαδή θα είναι ένας πίνακας δεικτών σε χαρακτήρα. Οι δείκτες του πίνακα που δεν χρησιμοποιούνται θα δείχνουν προφανώς σε NULL. Αντίστοιχα η εισαγωγή και η εξαγωγή κλειδιού στον πίνακα, γίνεται με τον ίδιο τρόπο που κάνουμε εισαγωγή και διαγραφή στοιχείου στις απλά συνδεδεμένες λίστες. Προφανώς δεν υπάρχει όριο στο πόσα στοιχεία θα περιέχει κάθε λίστα.

6.5.2. Αντιμετώπιση συγκρούσεων με ανοικτή διεύθυνση.

Λειτουργεί χωρίς την χρήση δεικτών. Τα κλειδιά αποθηκεύονται στον πίνακα ως εξής:

Με την χρήση της συνάρτησης κατακερματισμού υπολογίζουμε την θέση k του πίνακα, όπου θα φυλαχθεί το κλειδί.

- Αν η θέση k είναι κενή, τότε το κλειδί αποθηκεύεται εκεί.

- Αν η θέση k δεν είναι κενή, τότε δοκιμάζουμε μια θέση, η οποία προκύπτει από μια άλλη συνάρτηση, η οποία δέχεται ως ορίσματα την τιμή έναρξης και ένα βήμα.

α) Αντιμετώπιση συγκρούσεων με γραμμική δοκιμή.

Στην μέθοδο αυτή, εάν θέλουμε για παράδειγμα να κάνουμε εισαγωγή στον πίνακα κατακερματισμού των στοιχείων του συνόλου S = {37, 52, 43, 63, 9, 17, 70, 33, 72} χρησιμοποιούμε την συνάρτηση (h(k) + i) % m για 1 <= i<= n-1, όπου n το μέγεθος του πίνακα. Για m=11, τα στοιχεία θα τοποθετηθούν στον πίνακα όπως φαίνεται στο σχ. 6.9. Με κίτρινο χρώμα εμφανίζεται η θέση όπου θα έπρεπε να τοποθετηθεί το κλειδί, αλλά συμβαίνει σύγκρουση για πρώτη φορά και με πράσινο χρώμα εμφανίζεται η θέση όπου θα έπρεπε κατόπιν να τοποθετηθεί το κλειδί, αλλά συμβαίνει σύγκρουση για δεύτερη φορά:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
37					37						
52					37				52		
43					37				52		43
63					37				52	63	43
9	9				37				52	63	43
17	9				37		17		52	63	43
70	9				37	70	17		52	63	43
33	9	33			37	70	17		52	63	43
72	9	33			37	70	17	72	52	63	43

Σχ. 6.9

β) Αντιμετώπιση συγκρούσεων με τετραγωνική δοκιμή.

Η συνάρτηση κατακερματισμού είναι της μορφής:

$$(h_1(k) + c_1 * i + c_2 * i^2) \% m$$

Το i (i=0, 1, 2, ...) δηλώνει τις προσπάθειες για την εύρεση κενής θέσης και τα c_1 και c_2 είναι σταθερές. Εξετάζεται η πρώτη θέση για i = 0 και έπειτα αναζητούνται θέσεις σε αποστάσεις ανάλογες του τετραγώνου του i. Έχει μεγαλύτερη απόδοση από την γραμμική δοκιμή

Στην μέθοδο αυτή, εάν θέλουμε να κάνουμε εισαγωγή στον πίνακα κατακερματισμού των στοιχείων του συνόλου S = {37, 52, 43, 63, 9, 17, 70, 33,

72} με m=11, c_1=1 και c_2=2, τα στοιχεία θα τοποθετηθούν στον πίνακα όπως στο σχ. 6.10:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
37					37						
52					37				52		
43					37				52		43
63	63				37				52		43
9	63				37				52	9	43
17	63				37		17		52	9	43
70	63				37		17	70	52	9	43
33	63			33	37		17	70	52	9	43
72	63			33	37	72	17	70	52	9	43

Σχ. 6.10

ΚΕΦΑΛΑΙΟ 7
ΕΙΔΙΚΑ ΘΕΜΑΤΑ ΜΕΤΑΓΛΩΤΤΙΣΗΣ

7.1. ΠΑΡΑΜΕΤΡΟΙ ΓΡΑΜΜΗΣ ΕΝΤΟΛΩΝ.

Κατά την εκτέλεση ενός προγράμματος από την γραμμή εντολών, μπορούμε να στείλουμε σε αυτό κάποιες πληροφορίες. Αυτές αποτελούν τις παραμέτρους της γραμμής εντολών και είναι στην πραγματικότητα *παράμετροι τις οποίες μπορεί να δέχεται η συνάρτηση main( )*. Έστω για παράδειγμα ότι έχουμε γράψει ένα πρόγραμμα, του οποίου ο πηγαίος κώδικας έχει αποθηκευτεί σε ένα αρχείο με το όνομα code.c. Μετά την μεταγλώττιση έχει δημιουργηθεί το αρχείο code.exe, το οποίο ας πούμε ότι καλείται με δύο ορίσματα, π.χ. τα datain.txt και dataout.txt. Η κλήση ενός προγράμματος θα είναι κάτι όπως το παρακάτω:

code datain.txt dataout.txt

Η main( ) μπορεί να έχει μόνο δύο παραμέτρους και να έχει την εξής επικεφαλίδα:

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

Τα ονόματα των παραμέτρων, αν και δεν είναι απαραίτητα αυτά που αναφέρονται πιο πάνω, έχουν συνήθως τις τυπικές ονομασίες που βλέπετε και είναι ένας ακέραιος (ο argc) και ένας πίνακας δεικτών σε χαρακτήρα (ο argv). *Ο argc σημαίνει το πλήθος των ονομάτων της γραμμής εντολών μαζί με το όνομα του εκτελέσιμου αρχείου*. *Ο argv είναι πίνακας δεικτών σε χαρακτήρες. Κάθε δείκτης του πίνακα αυτού δείχνει στην αρχή μιας συμβολοσειράς, η οποία αποτελεί ένα όρισμα της γραμμής εντολών*. Στο παραπάνω παράδειγμα, το argc είναι ίσο με 3. Ο argv[0] δείχνει στην αρχή της συμβολοσειράς "code", ο argv[1] δείχνει στην αρχή της συμβολοσειράς "datain.txt", και ο argv[2] δείχνει στην αρχή της συμβολοσειράς "dataout.txt".

Στο παρακάτω πρόγραμμα υποτίθεται ότι ο χρήστης ζητείται να δώσει από το πληκτρολόγιο το όνομα χρήστη και τον κωδικό του, προκειμένου να εκτελεστεί το πρόγραμμα. Υποθέτοντας ότι το όνομα χρήστη είναι user και ο κωδικός είναι pswd, γίνεται έλεγχος για το εάν ο χρήστης έδωσε σωστό όνομα χρήστη και κωδικό από την γραμμή εντολών. Εάν το όνομα και ο κωδικός δεν είναι σωστά ή εάν δοθούν

περισσότερα ή λιγότερα από τα απαιτούμενα ορίσματα, στην οθόνη εμφανίζονται αντίστοιχα μηνύματα.

(Το παράδειγμα είναι από το βιβλίο «C Από τη θεωρία στην Εφαρμογή», Γ.Σ. Τσελίκης, Ν.Δ. Τσελίκας, ISBN: 978-960-93-1961-4).

```
int main (int argc, char *argv[ ]) {
   if (argc == 1)
        printf ("Error: missing user name and password\n");
   else
        if (argc == 2)
           printf ("Error: missing password\n");
        else
           if (argc == 3) {
                if (strcmp (argv[1], "user") == 0 && strcmp (argv[2], "pswd") ==0)
                   printf ("Valid user. The program ""%s"" will be executed...\n",
                              argv[0]);
                else
                   printf ("Wrong input\n"); }
           else
                printf ("Error: too many parameters\n");
   return 0; }
```

Στο περιβάλλον της Dev C++, οι παράμετροι της γραμμής εντολών δίνονται στην επιλογή Parameters της ετικέτας Execute.

7.2. ΠΡΟΓΡΑΜΜΑΤΑ ΜΕ ΠΟΛΛΑ ΑΡΧΕΙΑ.

Στον κλασικό μικρής κλίμακας προγραμματισμό, το πρόγραμμα που γράφουμε αποτελείται από ένα μόνο αρχείο. Στην πράξη, ένα πρόγραμμα στην C μπορεί να αποτελείται από περισσότερα από ένα αρχεία. Η πρακτική του χωρισμού σε πολλά αρχεία είναι ιδιαίτερα χρήσιμη, αφού βοηθάει τόσο στην καλύτερη δομή του προγράμματος, όσο και στον έλεγχο και την διόρθωση λαθών.

Θα δείξουμε την τεχνική μεταγλώττισης προγράμματος με πολλά αρχεία χρησιμοποιώντας το περιβάλλον της DevC++. Στο παράδειγμα που ακολουθεί, το πρόγραμμά μας περιέχει τρεις συναρτήσεις, τις message( ), display( ) και final( ):

```
#include <stdio.h>

void message ( );
void display (char *);
void final (char *);

int main(void) {
     message ( );
     display ("ARISTA");
     final ("ERGASIAS\n");
     return 0; }
```

```
void message ( ) {
    printf ("EPITYXIA!!!\n"); }

void display (char *grade) {
    printf ("BA8MOLOGEISTE ME %s!\n", grade); }

void final (char *msg) {
    printf ("TELOS %s\n", msg); }
```

Στο περιβάλλον της DEV C++ δημιουργούμε ένα Emptyproject για C με όνομα για παράδειγμα test. Δημιουργούμε τρία χωριστά αρχεία τα οποία προσθέτουμε στο test, χρησιμοποιώντας την επιλογή **Add to Project**. Τα τρία αυτά αρχεία είναι τα εξής:

α) Το αρχείο που θα περιέχει τις δηλώσεις των συναρτήσεων που θα χρησιμοποιήσουμε:

```
void message( );
void display(char *);
voidfinal(char *);
```

Το δημιουργούμε ως αρχείο κεφαλίδας (header, με προέκταση .h). Το ονομάζουμε για παράδειγμα **library.h** και θα βρίσκεται αποθηκευμένο στον τρέχοντα κατάλογο του προγράμματός μας.

β) Το αρχείο που θα περιέχει τους ορισμούς των συναρτήσεών μας:

```
void message ( ) {
    printf ("EPITYXIA!!!\n"); }

void display (char *grade) {
    printf ("BA8MOLOGEISTE ME %s!\n", grade); }

void final (char *msg) {
    printf("TELOS %s\n", msg); }
```

Συνηθίζουμε να του δίνουμε το ίδιο κύριο όνομα με το αρχείο κεφαλίδας, αλλά με προέκταση .c. Έτσι, το ονομάζουμε για παράδειγμα **library.c**.

γ) Το αρχείο που θα περιέχει την συνάρτηση main( ), (το ονομάζουμε π.χ.**main.c**):

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

#include "library.h"

int main(void) {
    message( );
    display ("ARISTA");
    final ("ERGASIAS\n");
    return 0; }
```

Παρατηρείστε ότι το δικό μας αρχείο βιβλιοθήκης library.h βρίσκεται μέσα σε " " και όχι μέσα σε <>, αναγκάζοντας έτσι τον compiler να το αναζητήσει μέσα στον ίδιο κατάλογο με αυτόν όπου βρίσκεται το αρχείο main.

7.3. ΕΞΩΤΕΡΙΚΕΣ ΜΕΤΑΒΛΗΤΕΣ ΣΕ ΠΡΟΓΡΑΜΜΑΤΑ ΜΕ ΠΟΛΛΑ ΑΡΧΕΙΑ.

Μια ερώτηση που προκύπτει είναι η εξής: εάν σε ένα πρόγραμμα με πολλά αρχεία δηλώσουμε μια εξωτερική μεταβλητή, αυτή είναι γνωστή και στα υπόλοιπα αρχεία; Ας δούμε μια παραλλαγή του προγράμματος της παραγράφου 7.2.:

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

void message( );
void display(char *);
void final(char *);

int ak;

int main(void) {
     scanf ("%d", &ak);
     message( );
     display("PHRATE ");
     final("ERGASIAS\n");
     return 0; }

void message( ) {
     if (ak >= 5)
          printf ("EPITYXIA!!!\n");
     else
          printf ("APOTYXATE\n"); }

void display(char *grade) {
     printf ("%s %d\n", grade, ak); }

void final(char *msg) {
     printf("TELOS %s\n", msg); }
```

Αν θελήσουμε τώρα να το «διασπάσουμε» σε πολλά αρχεία, θα έχουμε:

α) Για το αρχείο **library.h** ό,τι και στο προηγούμενο παράδειγμα, δηλαδή:

```
void message( );
void display(char *);
void final(char *);
```

β) Γιατοαρχείο**library.c**:

```
void message( ) {
   extern  int  ak;
   if (ak >= 5)
      printf ("EPITYXIA!!!\n");
   else
      printf ("APOTYXATE\n"); }

void display(char *grade) {
   extern  int  ak;
   printf ("%s %d\n", grade, ak); }

void final(char *msg) {
   printf("TELOS %s\n", msg); }
```

γ) Για το αρχείο **main.c**:

```
#include <stdio.h>
#include "library.h"
int ak;
int main(void) {
    scanf ("%d", &ak);
    message ( );
    display ("PHRATE");
    final ("ERGASIAS\n");
    return 0; }
```

Η δήλωση extern είναι απαραίτητη στις συναρτήσεις του library.c, αφού η ak είναι μεταβλητή δηλωμένη σε ένα άλλο αρχείο.

7.4. ΧΕΙΡΙΣΜΟΣ ΛΑΘΩΝ.

Η C προσφέρει διάφορα εργαλεία χειρισμού λαθών. Ένας συνηθισμένος και απλός τρόπος είναι για παράδειγμα ο έλεγχος μέσω μιας εντολής if της τιμής επιστροφής μιας συνάρτησης. Οι συναρτήσεις στην C πολλές φορές σε περίπτωση λάθους επιστρέφουν τιμή -1 ή NULL. Μέθοδοι όμως χειρισμού λαθών είναι οι παρακάτω:

7.4.1. Χρήση της μεταβλητής errno.

Όταν καλείται μια συνάρτηση στην C, μια μεταβλητή που λέγεται errno παίρνει τιμή ανάλογα με το είδος του λάθους που έχει συμβεί. Η errno είναι καθολική μεταβλητή και ορίζεται στο αρχείο κεφαλίδας errno.h. Μερικές τιμές της μεταβλητής και το είδος λάθους το οποίο σημαίνουν παρουσιάζονται στον πίνακα 7.1 που ακολουθεί παρακάτω.

Στο επόμενο παράδειγμα γίνεται προσπάθεια να «ανοίξει» για διάβασμα ένα αρχείο, το οποίο λέγεται keimeno.txt (θα μιλήσουμε σχετικά στο κεφάλαιο για τα αρχεία). Το αρχείο αυτό δεν υπάρχει, ο δείκτης fptrγίνεται ίσος με NULL και η πρώτη printf( )εμφανίσει μήνυμα λάθους 2.

Αντίστοιχα, η malloc αδυνατεί να δεσμεύσει τον χώρο που ζητείται, ο δείκτης ptrγίνεται ίσος με NULL και η δεύτερη printf( )εμφανίσει μήνυμα λάθους 12:

```
FILE * fptr;
char *ptr;
.....
errno=0 ;
.....
fptr = fopen("keimeno.txt", "r");
if (fptr == NULL)
     printf(" Value of errno: %d\n ", errno);
ptr = (char *) malloc (50000000000);
```

```
if (ptr == NULL)
    printf(" Value of errno: %d\n ", errno);
```

Τιμή	Είδος λάθους
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

Πίνακας 7.1.

7.4.2. Χρήση των συναρτήσεων perror( ) και strerror( ).

Είναι δυο συναρτήσεις, οι οποίες χρησιμοποιούνται για να δώσουν περιγραφή του λάθους που έχει συμβεί, εκτός από μόνο τον αριθμό του. Συγκεκριμένα:

α) Η συνάρτηση perror( ): Εμφανίζει μια συμβολοσειρά, την οποία της δώσαμε ως όρισμα ακολουθούμενη από άνω και κάτω τελεία, ένα κενό και το είδους λάθους. Δείτε το παράδειγμα της παραγράφου 7.4.1. τροποποιημένο λίγο:

```
FILE *fptr;
char *ptr;

fptr = fopen("ARXEIO.txt", "r");
if (fptr == NULL){
    printf("errno = %d\n", errno);
    perror("fopen"); }
ptr = (char *) malloc(50000000000);
if (ptr == NULL){
    printf("errno = %d\n", errno);
    perror("malloc"); }
```

Στην οθόνη θα εμφανιστεί:

```
errno = 2
fopen: No such file or directory
errno = 12
malloc: Notenoughspace
```

β) Η συνάρτηση strerror( ): Δέχεται ως όρισμα ένα ακέραιο, τον αριθμό λάθους και επιστρέφει ένα δείκτη σε χαρακτήρα, ο οποίος δείχνει στην αρχή μιας συμβολοσειράς όπου υπάρχει το μήνυμα λάθους. Ορίζεται στο string.h. Το πιο κάτω παράδειγμα:

```
FILE *fptr;
char *ptr;

fptr = fopen("ARXEIO.txt", "r");
if (fptr == NULL)
    printf("Error: %s\n", strerror(errno));
ptr = (char *) malloc(50000000000);
    if (ptr == NULL)
        printf("Error: %s\n", strerror(errno));
```

Στηνοθόνηθαεμφανίσει:

```
Error: No such file or directory
Error: Not enough space
```

7.4.3. Χρήση της συνάρτησης exit( ).

Στην C ορίζονται δύο σταθερές, οι EXIT_SUCCESS και EXIT_FAILURE, οι οποίες περνούν ως ορίσματα στην συνάρτηση exit( ), έτσι ώστε να δηλώσουν επιτυχή ή ανεπιτυχή τερματισμό ενός προγράμματος. Για παράδειγμα:

```
fptr = fopen("keimeno.txt", "r");
if (fptr == NULL) {
    printf (" Error: %s\n", strerror(errno));
    getchar ( );
    exit (EXIT_FAILURE); }
else {
    fclose (fptr);
    exit (EXIT_SUCCESS); }
```

Το παραπάνω πρόγραμμα τερματίζεται ανεπιτυχώς σε περίπτωση λάθους, αφού πρώτα εμφανίσει το μήνυμα **Error: No such file or directory** και αφού δοθεί ένας χαρακτήρας για την συνέχεια.

7.4.4. Διαίρεση με μηδέν.

Συχνό λάθος στην C είναι ο μη έλεγχος του παρονομαστή πριν την εκτέλεση μιας διαίρεσης. Δείτε στο παρακάτω ένα τρόπο επισήμανσης του λάθους:

```
float  fp, x;

if (fp == 0) {
     fprintf (stderr, "Division by zero!\n");
     getchar( );
     exit (EXIT_FAILURE); }
else {
     x = 1 / fp;
     printf ("%5.2f", x); }
```

7.4.5. Χρήση της μακροεντολής assert.

Η assert είναι μια μακροεντολή, η οποία χρησιμοποιείται για να ελέγξει συγκεκριμένες συνθήκες κατά τη διάρκεια της εκτέλεσης ενός προγράμματος, χρήσιμη για την αποσφαλμάτωση του προγράμματος. Χρειάζεται το αρχείο κεφαλίδας assert.h. Στην assert δίνεται ως όρισμα μια παράσταση. Εάν η παράσταση είναι αληθής, τότε η εκτέλεση του προγράμματος συνεχίζεται κανονικά. Εάν η παράσταση είναι ψευδής, το πρόγραμμα τερματίζεται και εμφανίζεται ένα μήνυμα στην οθόνη, το οποίο περιλαμβάνει το όνομα του αρχείου, τον αριθμό γραμμής και την συνθήκη η οποία δεν είναι αληθής.

Στο παράδειγμα που ακολουθεί διαβάζονται δύο ακέραιοι, ο a και ο b. Το πρόγραμμα κάνει την διαίρεση a/b, εκτός εάν δοθεί τιμή 0 στο b, οπότε το πρόγραμμα τερματίζεται:

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

int main( ) {
     int  a, b;

     printf ("Values for a and b\n");
     scanf ("%d%d", &a, &b);
     assert (b != 0);
     printf ("%d/%d = %.d\n", a, b, a/b);
     return 0; }
```

ΚΕΦΑΛΑΙΟ 8

ΕΙΣΟΔΟΣ - ΕΞΟΔΟΣ

8.1. ΓΕΝΙΚΑ.

Η είσοδος και έξοδος στην C γίνεται με την χρήση ροών. *Ροή* είναι ένα κανάλι μέσω του οποίου μια ακολουθία από bytes κυλούν, ρέουν δηλαδή, προς το πρόγραμμα ή από το πρόγραμμα. Μια *ροή εισόδου* έχει κατάληξη το πρόγραμμα, πρέπει όμως να γνωρίζουμε από πού πηγάζει, ενώ αντίστοιχα μια *ροή εξόδου* πηγάζει από το πρόγραμμα και πρέπει να γνωρίζουμε πού καταλήγει (π.χ. σε ένα αρχείο στον δίσκο, εκτυπωτή κλπ). Η έννοια των ροών ανεξαρτητοποιεί την σκέψη μας από συγκεκριμένες συσκευές. Οι ροές της C είναι δύο ειδών:

α) *Ροές κειμένου*. Λέγονται και *ροές υψηλού επιπέδου*. Εδώ η ροή αντιμετωπίζεται ως μια σειρά από χαρακτήρες και μιλάμε για «μορφοποιημένη είσοδο και έξοδο».

β) *Δυαδικές ροές*. Λέγονται και *ροές χαμηλού επιπέδου*. Εδώ η ροή αντιμετωπίζεται απλώς ως μια σειρά από byte, τα οποία διαβάζονται ή γράφονται χωρίς αναγκαστικά να αντιστοιχούν σε κείμενο.

8.2. ΠΡΟΚΑΘΟΡΙΣΜΕΝΕΣ ΡΟΕΣ ΤΗΣ C.

Οι παρακάτω τρεις προκαθορισμένες ροέςενεργοποιούνται μόλις αρχίσει να εκτελείται ένα πρόγραμμα σε C και κλείνουν με το τέλος του προγράμματος:

α) *Προκαθορισμένη ροή εισόδου*. Λέγεται **stdin** και συνδέεται με το πληκτρολόγιο.

β) *Προκαθορισμένη ροή εξόδου*. Λέγεται **stdout** και συνδέεται με την οθόνη.

γ) *Προκαθορισμένη ροή λάθους*. Λέγεται **stderr** και συνδέεται με την οθόνη.

Οι συναρτήσεις εισόδου – εξόδου της C χρησιμοποιούν είτε τις προκαθορισμένες ροές είτε χρειάζονται κάποια ροή ορισμένη από τον χρήστη, προκειμένου να υλοποιήσουν την είσοδο και την έξοδο στοιχείων. Στον παρακάτω πίνακα παρουσιάζονται τέτοιες συναρτήσεις:

Με προκαθορισμένη ροή	Με καθοριζόμενη από τον χρήστη ροή	Λειτουργία
printf	fprintf	Έξοδος δεδομένων
scanf	fscanf	Είσοδος δεδομένων
getchar	getc, fgetc	Είσοδος χαρακτήρα
putchar	putc, fputc	Έξοδος χαρακτήρα
puts	fputs	Έξοδος συμβολοσειράς
gets	fgets	Είσοδος συμβολοσειράς

Πίνακας 8.1.

8.3. ΕΙΣΟΔΟΣ.

8.3.1. Είσοδος χαρακτήρα.

(Η ταξινόμηση σε συναρτήσεις «ένταμιευμένες-μη ενταμιευμένες», σε «αντανακλώμενες στην stdin ή στην stdout» και μέρος των παραδειγμάτων που παρατίθενται έχουν ληφθεί από το βιβλίο «Εγχειρίδιο της C, PeterAitken&BradleyL. Jones, Εκδόσεις Μ. Γκιούρδας».)

Οι συναρτήσεις εισόδου χαρακτήρα διαβάζουν από μια ροή ένα χαρακτήρα κάθε φορά. Αν έχουμε φτάσει στο τέλος του αρχείου από το οποίο διαβάζουμε ή αν έχει συμβεί λάθος, επιστρέφουν τιμή EOF (EndOfFile), το οποίο είναι μια σταθερά ορισμένη στο stdio.h με τιμή ίση με -1. Οι συναρτήσεις αυτές μπορούν να ταξινομηθούν σε *ενταμιευμένες* (ή συναρτήσεις με ενδιάμεσο καταχωρητή) και σε *μη ενταμιευμένες*. *Ενταμιευμένες* είναι εκείνες οι συναρτήσεις εισόδου, στις οποίες το λειτουργικό σύστημα αποθηκεύει τους χαρακτήρες σε ένα προσωρινό καταχωρητή μέχρι να πατηθεί το Enter. Μετά το πάτημα του Enter, οι χαρακτήρες στέλνονται στην ροή stdin. Στις *μη ενταμιευμένες* συναρτήσεις κάθε χαρακτήρας στέλνεται στην ροή stdin μόλις αυτός δοθεί από το πληκτρολόγιο.

Μια άλλη ταξινόμηση των συναρτήσεων εισόδου χαρακτήρα τις διακρίνει σε αυτές που *αντανακλούν* κάθε χαρακτήρα εισόδου στην stdout, άρα στην οθόνη και σε αυτές που *δεν αντανακλούν*.

α) Η συνάρτηση **getchar( )**:

Είναι *ενταμιευμένη συνάρτηση με αντανάκλαση*. Στο παρακάτω, καλείται συνεχώς η συνάρτηση getchar( ) μέχρι να δώσετε <Enter>. Οι χαρακτήρες εμφανίζονται στην οθόνη μετά το πάτημα του <Enter>, όμως αντανακλώνται άμεσα στην οθόνη κατά την πληκτρολόγησή τους

```
char ch;
.....
while ((ch = getchar( )) != '\n')
    putchar(ch);
```

Αν πληκτρολογήσω το A και enter, αυτό θα εμφανιστεί στην οθόνη. Αν δώσω qwerty και μετά enter, στην οθόνη θα εμφανιστεί qwerty, γιατί οι τιμές που διαβάστηκαν μεταφέρθηκαν στον ενδιάμεσο καταχωρητή και όταν δόθηκε το enter στάλθηκαν στην stdout. Οι χαρακτήρες που πληκτρολογώ φαίνονται στην οθόνη, διότι η getchar( ) είναι συνάρτηση με αντανάκλαση.

β) Η συνάρτηση **getche( )** και η συνάρτηση **getch( )**:

Είναι *μη ενταμιευμένες συναρτήσεις*. *Η πρώτη είναι συνάρτηση με αντανάκλαση, ενώ η δεύτερη είναι χωρίς αντανάκλαση*. Πρέπει να επισημανθεί παρά την μεγάλη τους χρησιμότητα, ότι δεν είναι συναρτήσεις της ANSIC και για τον λόγο αυτό πρέπει να χρησιμοποιούνται με προσοχή αν θέλουμε φορητότητα των προγραμμάτων.

γ) Η συνάρτηση **getc( )** και η συνάρτηση **fgetc( )**:

Θα μιλήσουμε γι'αυτές στην ενότητα των αρχείων της C.

8.3.2. Είσοδος συμβολοσειράς.

Οι συναρτήσεις εδώ διαβάζουν μια ολόκληρη γραμμή από μια ροή εισόδου μέχρι να δοθεί <Enter>.

α) Η συνάρτηση **gets( )**:

Διαβάζει μια γραμμή από την stdin μέχρι να συναντήσει χαρακτήρα \n ή χαρακτήρα τέλους αρχείου. Αποθηκεύει σε ένα πίνακα τύπου char και δημιουργεί έτσι μια συμβολοσειρά.

β) Η συνάρτηση **fgets( )**:

Στην συνάρτηση fgets( ) θα αναφερθούμε στο κεφάλαιο χειρισμού αρχείων στην C. Παρουσιάζουμε πάντως ένα παράδειγμα χρήσης της fgets για την προκαθορισμένη ροή εισόδου:

```
#define N 10
int main( ) {
    char pin[N];
    while (1) {
        fgets(pin, N, stdin);
        if (pin[0] == '\n')
            break;
        puts(pin); }
    .....
```

Το πρόγραμμα θα διαβάζει συνεχώς συμβολοσειρές από το πληκτρολόγιο μέχρι να δοθεί κενή συμβολοσειρά.

8.3.3. Μορφοποιημένη είσοδος.

Αναφερόμαστε σε συναρτήσεις μορφοποίησης της εισόδου, δηλαδή συναρτήσεις στις οποίες καθορίζουμε τι είδους τιμές θα εισαγάγουμε, πού θα αποθηκευτούν κλπ. Τέτοιες συναρτήσεις είναι η scanf( ) και η fscanf( ), την οποία θα συναντήσουμε στο κεφάλαιο για τα αρχεία στην C.

Η scanf( ) παρέχει ενταμιευμένη είσοδο. Αυτό δημιουργεί κάποια προβλήματα στην χρήση της scanf( ), τα οποία ο χρήστης πρέπει να ξέρει να τα αντιμετωπίσει. Θυμίζουμε ότι στην ενταμιευμένη είσοδο το λειτουργικό σύστημα αποθηκεύει τους χαρακτήρες σε ένα προσωρινό καταχωρητή μέχρι να πατηθεί το Enter. Ό,τι αποθηκεύεται εκεί υφίσταται επεξεργασία από την scanf( ), η εκτέλεση του προγράμματος συνεχίζεται όταν έχει ληφθεί η πληροφορία που χρειάζεται ώστε να καλύπτονται τα καθοριζόμενα στο τμήμα της μορφοποίησης της scanf( ), δηλαδή μέσα στα διπλά εισαγωγικά. Έτσι, στα παραδείγματα που ακολουθούν παρουσιάζονται προβλήματα τα οποία μπορεί να ανακύψουν από μη σωστή χρήση της scanf( ):

Εκτελούμε την παρακάτω scanf( ):

```
scanf ("%d%d", &ak, &ms);
```

Η scanf( ) περιμένει δυο ακεραίους.

α) Αν πληκτρολογήσουμε π.χ. 20υ30 και μετά <Enter>, δεν υπάρχει κανένα πρόβλημα και η scanf( ) εκτελείται κανονικά. Όλοι οι χαρακτήρες από την stdin έχουν «απορροφηθεί» από την scanf( ). Αν πληκτρολογήσουμε λιγότερους από δύο ακεραίους, η scanf( ) συνεχίζει να περιμένει μέχρι να δοθεί και ο άλλος ακέραιος.

β) Έστω το παρακάτω πρόγραμμα:

```
scanf ("%d%d", &ak, &ms);
printf ("%d %d\n", ak, ms);
scanf ("%d%d", &ak, &ms);
printf ("%d %d\n", ak, ms);
```

Αν πληκτρολογήσουμε π.χ. 20υ30υ15 (δηλαδή περισσότερους ακέραιους από τους αναμενόμενους) και μετά <Enter>, η scanf( ) θα διαβάσει το 20 και το 30 και θα τελειώσει την εργασία της, οπότε στην οθόνη θα εμφανιστεί 20υ30. Τα 1 και 5 παραμένουν στην ροή stdin. Αν αμέσως μετά διαβάσουμε ξανά δυο ακέραιους, π.χ. 7υ6 και μετά <Enter>, στην οθόνη θα εμφανιστεί: 15υ7.

Η παραμονή τέτοιων «υπολοίπων» στην stdin έχει ως συνέπεια την εμφάνιση προβλημάτων κατά το διάβασμα χαρακτήρα ή συμβολοσειράς μετά από κάτι άλλο, π.χ. ένα ακέραιο. Αυτό διότι, μετά το διάβασμα του ακεραίου στο stdin παραμένει το <Enter>, το οποίο θεωρείται ότι αποτελεί τον χαρακτήρα ή την συμβολοσειρά που αναμένεται.

Για να αποφύγουμε προβλήματα όπως τα παραπάνω, πριν το διάβασμα χαρακτήρα ή συμβολοσειράς, συνιστάται η χρήση της συνάρτησης fflush( ), η

οποία αδειάζει μια ροή, άρα φυσικά και την προκαθορισμένη ροή εισόδου. Η σύνταξή της εν προκειμένω θα είναι:

fflush (stdin);

8.4. ΕΞΟΔΟΣ.

8.4.1. Έξοδος χαρακτήρα.

α) Η συνάρτηση **putchar( )**: Στέλνει στην stdout το όρισμά της. Επιστρέφει τον χαρακτήρα που γράφτηκε ή EOF αν υπάρξει λάθος

β) Η συνάρτηση **putc( )** ή**fputc( )**: Θα μιλήσουμε γι'αυτές στην ενότητα των αρχείων της C.

8.4.2. Έξοδος συμβολοσειράς.

α) Η συνάρτηση **puts( )**: Στέλνει στην stdout το όρισμά της, προσθέτοντας και ένα χαρακτήρα νέας γραμμής.

β) Η συνάρτηση **fputs( )**: Θα μιλήσουμε γι'αυτήν στην ενότητα αρχείων της C.

8.4.3. Μορφοποιημένη έξοδος.

α) Η συνάρτηση **printf( )**: Εδώ καθορίζουμε τι είδους τιμές θα γράψουμε, πώς θα γραφούν κλπ. Στέλνει τις τιμές στην stdout.

β) Η συνάρτηση **fprintf( )**: Θα μιλήσουμε γι'αυτήν στην ενότητα των αρχείων της C.

ΚΕΦΑΛΑΙΟ 9

ΑΡΧΕΙΑ

9.1. ΓΕΝΙΚΑ ΓΙΑ ΤΑ ΑΡΧΕΙΑ.

Αρχείο είναι ένα σύνολο από byte, τα οποία αποθηκεύονται σε μια μονάδα περιφερειακής μνήμης, δεν αναφερόμαστε δηλαδή στην κύρια μνήμη του υπολογιστή. Η C χειρίζεται δύο είδη αρχείων:

- Τα **αρχεία κειμένου** (text files). Σε αυτό το είδος των αρχείων τα δεδομένα αποθηκεύονται με την ίδια μορφή, με την οποία θα τα βλέπαμε και στην οθόνη. Αν για παράδειγμα αποθηκεύσουμε τον αριθμό 53, θα αποθηκευτεί το ψηφίο 5 και το ψηφίο 3, άρα δύο byte συνολικά. Το τέλος ενός αρχείου κειμένου σηματοδοτείται από την ύπαρξη ενός **ειδικού χαρακτήρα**, ο οποίος λέγεται **EOF** (End Of File). Όταν γράφουμε σε ένα αρχείο κειμένου, ο χαρακτήρας αλλαγής γραμμής (το \n δηλαδή) μετατρέπεται σε δύο χαρακτήρες, στον χαρακτήρα τροφοδοσίας γραμμής (Line Feed, LF), ο οποίος στον κώδικα ASCII έχει αριθμό 10 και στον χαρακτήρα επαναφοράς κεφαλής (Carrier Return, CR) με κωδικό 13 στον ASCII. (Η ορολογία «τροφοδοσία γραμμής» και «επαναφορά κεφαλής» μας έρχεται από τις παλιές γραφομηχανές).
- Τα **δυαδικά αρχεία** (binary files). Σε αυτά, τα δεδομένα αποθηκεύονται δυαδικά, δηλαδή αποθηκεύονται με BCD μορφή. Στην αντίστοιχη με την περίπτωση που αναφέρθηκε πιο πάνω, ο αριθμός 53 θα αποθηκευτεί τώρα με 4 byte, τα οποία είναι τα byte που απεικονίζουν τον συγκεκριμένο ακέραιο. Στα αρχεία αυτά δεν υπάρχει κάποιος ειδικός χαρακτήρας, ο οποίος να συμβολίζει το τέλος του αρχείου.

9.2. ΠΡΟΣΠΕΛΑΣΗ ΑΡΧΕΙΩΝ ΓΕΝΙΚΑ.

Για να μπορέσουμε να χειριστούμε ένα αρχείο στην C, πρέπει πρώτα *να δημιουργήσουμε μια ροή εισόδου-εξόδου,* την οποία να συνδέσουμε στο αρχείο. Μέσω της ροής αυτής γίνεται η επεξεργασία του αρχείου, δηλαδή η διακίνηση

πληροφοριών από και προς αυτό, ενώ όταν τελειώσουμε την επεξεργασία, πρέπει να αποσυνδέσουμε την ροή από το αρχείο. Η δημιουργία της ροής γίνεται με την δήλωση ενός **δείκτη τύπου FILE**, όπως παρακάτω:

FILE *fptr;

Ακολουθεί η σύνδεση του ρεύματος που δημιουργήθηκε με το αρχείο. Η σύνδεση γίνεται με την κλήση της συνάρτησης **fopen( )**. Η σύνταξή της είναι:

fptr = fopen ("Όνομα αρχείου", "Λειτουργία");

Στην παραπάνω εντολή:

- Το **"Όνομα αρχείου"** είναι ένα σύνολο από χαρακτήρες (μια συμβολοσειρά), το οποίο προσδιορίζει το όνομα και την διαδρομή (path) του αρχείου. Στην πραγματικότητα είναι ένας δείκτης σε χαρακτήρα, ο οποίος δείχνει στην αρχή αυτής της διαδρομής.
- Η **"Λειτουργία"** είναι ένα σύνολο από χαρακτήρες (μια συμβολοσειρά), το οποίο προσδιορίζει το τι ενέργεια θα γίνει στο συγκεκριμένο αρχείο.

Στο παράδειγμα που ακολουθεί ανοίγουμε για διάβασμα ένα αρχείο κειμένου που λέγεται points.txt και βρίσκεται στον κατάλογο data του δίσκου C:

```
FILE *fptr;
fptr = fopen ("C:/data/points.txt", "r");
```

Σε αδυναμία ανοίγματος του αρχείου, η fopen( ) επιστρέφει NULL. Αδυναμία ανοίγματος μπορεί να προκύψει από διάφορους λόγους, όπως π.χ. από χρήση μη έγκυρου ονόματος αρχείου, από προσπάθεια ανοίγματος για διάβασμα ενός ανύπαρκτου αρχείου κλπ.

Σημείωση:Για τον καθορισμό της διαδρομής όπου βρίσκεται το αρχείο χρησιμοποιείται η κάθετος (/) και όχι η ανάποδη κάθετος (\).

Η δεύτερη συμβολοσειρά στην κλήση της fopen( ) καθορίζει όπως αναφέραμε την ενέργεια, η οποία θα γίνει στο συγκεκριμένο αρχείο. Στον Πίνακα 9.1 που ακολουθεί παρουσιάζονται τα εξής:

- Στην πρώτη και δεύτερη στήλη η συμβολοσειρά που καθορίζει την λειτουργία.
- Στην τρίτη στήλη η ενέργεια που θα γίνει στο αρχείο.
- Στην τέταρτη στήλη η θέση του αρχείου, στην οποία τοποθετείται ο δείκτης θέσης αρχείου, τον οποίο αναφέραμε προηγουμένως, μετά την εκτέλεση της εντολής.

Ο τύπος FILE είναι ορισμένος στο stdio.h. Η fopen( ) δημιουργεί και ένα προσωρινό καταχωρητή. Ο δείκτης fptr δεν δείχνει το πραγματικό αρχείο, αλλά μια δομή,η οποία περιέχει πληροφορίες σχετικές με το αρχείο που διαβάζεται και τον προσωρινό

καταχωρητή. Μεταξύ των άλλων πεδίων της περιέχει και ένα πεδίο δείκτη (**δείκτης θέσης**), το οποίο δείχνει σε ποια θέση του αρχείου θα γίνει η επόμενη λειτουργία εγγραφής ή ανάγνωσης. Αν δεν μπορέσει να ανοίξει το αρχείο, η συνάρτηση fopen() επιστρέφει την τιμή NULL, οπότε σε συνδυασμό με την if και την exit() σταματά το πρόγραμμα.

Μετά από κάθε τέτοια λειτουργία, ο δείκτης θέσης ενημερώνεται αυτόματα, μετακινείται κατά μια θέση μέσα στο αρχείο και δείχνει στο επόμενο byte.

Μετά την επεξεργασία ενός αρχείου, *πρέπει να «κλείνει» το αρχείο,* να παύει δηλαδή η δυνατότητα προσπέλασης σε αυτό. Το παραπάνω γίνεται με την κλήση της συνάρτησης **fclose( )**, η οποία αποκόπτει την ροή από τον δείκτη αρχείου:

fclose (fptr);

Λειτουργία για αρχεία κειμένου	**Λειτουργία για δυαδικά αρχεία**	**Ενέργεια**	**Θέση δείκτη**	**Παρατηρήσεις όσον αφορά το αρχείο**
"r"	"rb"	Διάβασμα	Αρχή	Πρέπει να υπάρχει.
"w"	"wb"	Εγγραφή	Αρχή	Αν ήδη υπάρχει, διαγράφονται τα περιεχόμενά του. Αν δεν υπάρχει, δημιουργείται.
"a"	"ab"	Εγγραφή	Τέλος	Τοποθέτηση δεδομένων μετά το τέλος του. Αν δεν υπάρχει, δημιουργείται.
"r+"	"rb+"	Διάβασμα και εγγραφή	Αρχή	Πρέπει να υπάρχει.
"w+"	"wb+"	Διάβασμα και εγγραφή	Αρχή	Αν ήδη υπάρχει, διαγράφονται τα περιεχόμενά του. Αν δεν υπάρχει, δημιουργείται.
"a+"	"ab+"	Διάβασμα και εγγραφή	Αρχή για διάβασμα, τέλος για εγγραφή	Τοποθέτηση δεδομένων μετά το τέλος του. Αν δεν υπάρχει, δημιουργείται.

Πίνακας 9.1. Τύποι προσπέλασης αρχείων.

Μεταξύ ενός αρχείου και μιας περιφερειακής συσκευής παρεμβάλλεται ένας *χώρος ενδιάμεσης αποθήκευσηςδεδομένων* (buffer). Στον χώρο αυτόν αποθηκεύονται τα δεδομένα κατά την μεταφορά τους από και προς το αρχείο. Για κάθε ροή δημιουργείται ο δικός της χώρος ενδιάμεσηςαποθήκευσης. Όταν γεμίσει αυτός ο χώρος, εκκενώνεται αυτόματα, μπορούμε όμως να προβούμε και σε εξαναγκασμένη εκκένωσή του. Κατά την εκτέλεση της συνάρτησης fclose( ), πριν κλείσει το αρχείο, μεταφέρονται σε αυτό όσα υπάρχουν στον χώρο ενδιάμεσηςαποθήκευσης.

Εξαναγκασμένη εκκένωση του χώρου που συνδέεται με το αρχείο που δείχνει ο δείκτης fptr, γίνεται με την κλήση της συνάρτησης fflush( ), η οποία συντάσσεται ως εξής:

fflush (fptr);

9.3. ΠΡΟΣΠΕΛΑΣΗ ΑΡΧΕΙΩΝ ΚΕΙΜΕΝΟΥ.

Συναρτήσεις που διέπουν τα αρχεία κειμένου είναι οι:

9.3.1. fputc( ). *Γράφει ένα χαρακτήρα σε ένα αρχείο.* Έχει τιμή επιστροφής τον αύξοντα αριθμό σε ASCII του χαρακτήρα που έγραψε στο αρχείο ή EOF εάν δεν γράφτηκε ο χαρακτήρας στο αρχείο. Η δήλωσή της είναι:

int fputc (int, FILE*);

Ισοδύναμη είναι και η **putc( )**, η οποία υλοποιείται ως μακροεντολή και συνεπώς είναι ταχύτερη από την fputc( ).

Στο παρακάτω πρόγραμμα διαβάζονται συνεχώς χαρακτήρες από το πλήκτρολόγιο μέχρι να δοθεί ο χαρακτήρας τελεία (.) και αποθηκεύονται στο αρχείο points.txt:

```
FILE  *fptr;
char  ch;

fptr = fopen ("C:/data/points.txt", "w");
if (fptr == NULL)
      exit (EXIT_FAILURE);
ch = getche( );
while (ch != '.') {
     fputc (ch, fptr);
     ch = getche( ); }
fclose (fptr);
```

Αν στο παραπάνω πρόγραμμα αντικαταστήσουμε την fputc(ch, fptr) με την **fputc (ch, stdout);**το πρόγραμμα προφανώς εμφανίζει τους χαρακτήρες στην οθόνη.

9.3.2. fgetc( ). *Διαβάζει ένα χαρακτήρα από ένα αρχείο.* Έχει τιμή επιστροφής τον αύξοντα αριθμό σε ASCII του χαρακτήρα που διάβασε ή EOF εάν φτάσει το διάβασμα στο τέλος του αρχείου. Η δήλωσή της είναι:

int fgetc (FILE*);

Αντίστοιχα με την putc( ) υπάρχει και η συνάρτηση **getc( )**, η οποία υλοποιείται ως μακροεντολή.

Στο πρόγραμμα που ακολουθεί διαβάζονται συνεχώς χαρακτήρες από το αρχείο points.txt μέχρι να διαβαστούν όλοι οι χαρακτήρες του αρχείου και ένας-ένας γράφονται στην οθόνη. Μετά από κάθε διάβασμα έχουμε και εδώ μετακίνηση του δείκτη θέσης του αρχείου κατά μια θέση:

```
FILE  *fptr;
char  ch;

fptr = fopen ("C:/data/points.txt", "r");
if (fptr == NULL)
     exit (EXIT_FAILURE);
ch = fgetc(fptr);
putchar(ch);
while (ch != EOF) {
     ch = fgetc (fptr);
     putchar(ch); }
fclose (fptr);
```

Το παρακάτω πρόγραμμα αποτελεί μια μικρή τροποποίηση του προηγουμένου. Εδώ και πάλι διαβάζονται όλοι οι χαρακτήρες από ένα αρχείο και εμφανίζονται ένας-ένας στην οθόνη, αλλά το αρχείο δίνεται ως παράμετρος της main( ) από την γραμμή εντολών. Το πρόγραμμα διαπιστώνει εάν δόθηκε η παράμετρος αυτή ελέγχοντας την τιμή της μεταβλητής argc:

```
main (int argc, char *argv[ ]) {
     FILE  *fptr;
     char ch;

     if (argc!=2) {
          printf("Filename is missing\n");
          exit(EXIT_FAILURE); }

     if ( (fptr=fopen (argv [1], "r") )==NULL) {
          printf("Error: %s\n", strerror(errno));
          exit(EXIT_FAILURE); }

     ch=fgetc(fptr);
     while (ch!= EOF) {
          putchar (ch);
          ch=fgetc(fptr); }

     fclose (fptr); }
```

Προς αποφυγή λαθών, η τιμή επιστροφής των fgetc( ) και getc( ) θα πρέπει να αποθηκεύεται σε τιμές τύπου int και όχι char. Δείτε τι θα συμβεί εάν κατά την εκτέλεση του πιο κάτω προγράμματος δώσουμε από το πληκτρολόγιο κατά σειρά τους χαρακτήρες a b c d $ e f g $

```
FILE  *fptr;
char ch;
```

```
fptr = fopen ("C:/data/points.txt", "w");

ch=getche( );
while (ch != '$') {
      putc(ch, fptr);
      ch=getche(); }

ch = 255;

while (ch != '$') {
      putc(ch, fptr);
      ch=getche(); }

fclose (fptr);

fptr = fopen ("C:/data/points.txt", "r");

ch = getc(fptr);
while (ch != EOF) {
      putchar(ch);
      ch = fgetc (fptr); }

fclose (fptr);
```

Στην οθόνη θα πάρουμε:

a b c d

Αυτό διότι το 255 το πρόγραμμά μας το αντιλαμβάνεται ως -1, δεδομένου ότι είναι ίσο με 10000000. Άρα, βλέποντας μια τιμή του χαρακτήρα ch ίση με -1, καταλαβαίνει ότι πρόκειται για τον χαρακτήρα EOF και το διάβασμα του αρχείου τερματίζεται. Κάτι τέτοιο δεν θα συμβεί εάν το ch έχει δηλωθεί ως ακέραιος.

9.3.3. fprintf( ). *Γράφει σε ένα αρχείο.* Η σύνταξή της είναι παρόμοια με της printf( ), με τη διαφορά ότι γράφει στο αρχείο και όχι στην οθόνη, άρα καθορίζεται και ροή προς το αρχείο. Είναι συνάρτηση μορφοποιημένης εξόδου, μας δίνει δηλαδή τη δυνατότητα να αποθηκεύσουμε τα δεδομένα που θέλουμε (όχι μόνο χαρακτήρες) και με τη μορφή που θέλουμε.

Στο παρακάτω πρόγραμμα διαβάζονται συνεχώς ακέραιοι από το πληκτρολόγιο, μέχρι να δοθεί μηδέν. Οι ακέραιοι αυτοί γράφονται στο αρχείο akeraioi.txt. Για κάθε εγγραφή στο αρχείο, ο δείκτης θέσης αρχείου προχωράει κατά 4. Παρατηρείστε την σύνταξη της fprintf( ):

```
FILE  *fptr;
int  ak;

fptr = fopen ("C:/data/akeraioi.txt", "w");
scanf ("%d", &ak);
while (ak != 0) {
      fprintf (fptr, "%d\n", ak);
```

```
        scanf ("%d", &ak); }
    fclose (fptr);
```

Προφανώς, εάν αντικατασταθεί το fptr με stdout, η fprintf( ) γίνεται ισοδύναμη με την printf( ).

Η τιμή επιστροφής της fprintf( ) ισούται με τον αριθμό των χαρακτήρων που γράφτηκαν στο αρχείο, αλλιώς EOF, άρα το παρακάτω πρόγραμμα θα γράψει στην οθόνη τον αριθμό 5:

```
FILE  *fptr;
int  ak;

fptr = fopen ("C:/data/akeraioi.txt", "w");
if (fptr != NULL)
     ak = fprintf (fptr, "ΕΝΝΙΑ");
printf ("%d\n", ak);
fclose (fptr);
```

Ας δούμε για άσκηση μερικά άλλα προγράμματα που χρησιμοποιούν την fprintf( ).

α) Το πρόγραμμα που ακολουθεί θα γράψει στο αρχείο το ΕΝΑ και στην οθόνη το -1 (δηλαδή το EOF):

```
FILE  *fptr[2];

if ((fptr[0] = fptr[1] = fopen ("data.txt", "w")) != NULL) {
     fprintf (fptr[0], "ENA");
     fclose (fptr[0]);
     printf ("%d\n", fprintf(fptr[1], "ΔΥΟ"));
     fclose (fptr[1]); }
```

β) Στο παρακάτω πρόγραμμα θέλουμε να ανοίγει το αρχείο η συνάρτηση open_file( ). Στο πρόγραμμα υπάρχει λάθος:

```
void open_file(char [ ], FILE *);

main( ){
     FILE  *fptr=NULL;

     open_file ("data.txt ", fprt);
     fprintf (fptr, "SUCCESS\n");
     fclose (fptr); }

void open_file(char filename[ ], FILE *fpo) {
     fpo = fopen(filename, "w"); }
```

Συγκεκριμένα, η συνάρτηση πρέπει να επιστρέφει τον δείκτη σε FILE για το άνοιγμα του αρχείου, άρα το σωστό πρόγραμμα θα είναι:

```
void open_file(char [ ], FILE **);
main( ){
    FILE  *fptr=NULL;
    open_file("data.txt ", &fprt);
    fprintf (fptr, "SUCCESS\n");
    fclose (fptr); }

void open_file(char filename[ ], FILE **fpo) {
    *fpo = fopen(filename, "w"); }
```

9.3.4. fscanf( ). *Διαβάζει από αρχείο.* Η σύνταξή της είναι σχεδόν όπως και της scanf(), με τη διαφορά ότι διαβάζει από το αρχείο και όχι το πληκτρολόγιο, άρα καθορίζεται και ροή από το αρχείο.

Το παρακάτω πρόγραμμα διαβάζει συνεχώς ακέραιους από το αρχείο akeraioi.txt, μέχρι να συναντήσει θετικό αριθμό. Οι ακέραιοι αυτοί γράφονται στην οθόνη. Παρατηρείστε την σύνταξη της fscanf( ):

```
FILE  *fptr;
int  ak;
fptr = fopen ("C:/data/akeraioi.txt", "r");
fscanf (fptr, "%d", &ak);
while (ak < 0) {
     printf ("%d\n", ak);
     fscanf (fptr, "%d", &ak); }
fclose (fptr);
```

Για να χρησιμοποιήσουμε το EOF, το οποίο αναφέραμε προηγουμένως, πρέπει να διαβάζουμε το αρχείο χαρακτήρα προς χαρακτήρα. Τα παραπάνω δεν ισχύουν εάν το αρχείο περιέχει μορφοποιημένα δεδομένα, όπως αυτά που γράφτηκαν με την χρήση της fprintf( ). Αν θέλουμε τότε να διαπιστώσουμε εάν έχουμε φτάσει στο τέλος του αρχείου, χρησιμοποιούμε την συνάρτηση feof( ), η οποία παρουσιάζεται πιο κάτω.

Η χρήση της fscanf( ) προϋποθέτει ότι είναι γνωστοί ο τύπος και ο τρόπος αποθήκευσης των δεδομένων. Για παράδειγμα, στο παρακάτω:

```
fscanf (fptr, "%s%d%f", name, &k, &fs);
```

Θα διαβάσει τιμές για μια συμβολοσειρά, ένα ακέραιο και ένα float, άρα με αυτό τον τρόπο πρέπει να έχει γίνει και η αποθήκευση στο αρχείο.Δείτε και το παρακάτω παράδειγμα:

Σε ένα αρχείο, το data/xwres.txt, υποτίθεται ότι είναι αποθηκευμένα σε εγγραφές (structures) τα στοιχεία num χωρών, με τις εξής πληροφορίες για κάθε χώρα: το όνομα της χώρας, η πρωτεύουσά της και ο πληθυσμός σε μια γραμμή για κάθε εγγραφή. Διαβάζουμε από το αρχείο και αποθηκεύουμε στη μνήμη τα στοιχεία κάθε χώρας. Επιπλέον, δίνουμε ένα ακέραιο, τον atoma, ο οποίος αντιπροσωπεύει πληθυσμό. Με το παρακάτω πρόγραμμα εμφανίζονται στην οθόνη οι χώρες και οι πρωτεύουσές τους εάν ο πληθυσμός είναι μεγαλύτερος ή ίσος με το atoma:

```
typedef struct xwra {
      char name[30];
      char capital[30];
      int popul; } XWRA;

int main( ) {
      FILE *fptr;
      XWRA *ptr;
      int k, num, atoma;

      scanf ("%d", &num);
      fptr = fopen ("C:/data/xwres.txt", "r");
      ptr = (XWRA *) malloc (num * sizeof(XWRA));
      scanf ("%d", &atoma);

      for (k=0; k<num; k++) {
            fscanf (fptr, "%s%s%d", (ptr+k)->name, (ptr+k)->capital,
                  &(ptr+k)->popul);
            if ((ptr+k)->popul >= atoma)
                  printf ("%s%sd\n", (ptr+k)->name, (ptr+k)->capital); }
      fclose (fptr);
      return 0; }
```

9.3.5.feof( ). Έχει τιμή επιστροφής διάφορη του μηδενός όταν κάποια συνάρτηση επιχειρεί να διαβάσει πέρα από το σημείο όπου τερματίζονται τα δεδομένα ενός αρχείου και μηδέν σε κάθε άλλη περίπτωση. Δέχεται ένα όρισμα, το οποίο καθορίζει την ροή προς και από το αρχείο. Η δήλωσή της είναι:

int feof (FILE*);

Το επόμενο παράδειγμα αποτελεί παραλλαγή αυτού της παραγράφου 9.3.4. Εδώ διαβάζουμε ακέραιους από το αρχείο μέχρι να φτάσουμε στο τέλος του. Τους ακέραιους αυτούς τους εμφανίζουμε στην οθόνη:

FILE *fptr;

```
int  ak;

fptr = fopen ("C:/data/akeraioi.txt", "r");
fscanf (fptr, "%d", &ak);
while (! feof (fptr)) {
     printf ("%d\n", ak);
     fscanf (fptr, "%d", &ak); }
fclose (fptr);
```

Αντίστοιχα, στο παρακάτω πρόγραμμα διαβάζουμε από το αρχείο το πολύ 100 ακέραιους, τους οποίους εμφανίζουμε στην οθόνη. Εάν το αρχείο περιέχει λιγότερους αριθμούς, τότε θα διαβάζει μόνο όσους υπάρχουν. Ο έλεγχος του δείκτη fptr προφανώς πρέπει να γίνεται σε κάθε πρόγραμμα (και στο προηγούμενο φυσικά):

```
FILE  *fptr;
int  ak, k;

fptr = fopen ("C:/data/akeraioi.txt", "r");
if (fptr != NULL) {
     for (k=1; k<=100; k++) {
          fscanf (fptr, "%d", &ak);
          printf ("%d\n", ak);
          if (feof (fptr))
               break; } }
fclose (fptr);
```

Ας δούμε ένα παράδειγμα προβλήματος από το οποίο μας απαλλάσσει η χρήση της feof( ). Ας θεωρήσουμε ότι χρησιμοποιήσαμε ένα συνδυασμό δύο προηγουμένων προγραμμάτων για το γράψιμο σε ένα αρχείο και το διάβασμα από αυτό. Συγκεκριμένα:

```
FILE  *fptr;
int  ak;

fptr = fopen ("C:/data/akeraioi.txt", "w");
scanf ("%d", &ak);
while (ak != 0) {
     fprintf (fptr, "%d\n", ak);
     scanf ("%d", &ak); }
fclose (fptr);

fptr = fopen ("C:/data/akeraioi.txt", "r");
fscanf (fptr, "%d", &ak);
while (ak != 0) {
     printf ("%d\n", ak);
     fscanf (fptr, "%d", &ak); }
fclose (fptr);
```

και έστω ότι από το πληκτρολόγιο δίνω τους ακεραίους 1, 2, 3, 4, 5 και 0. Το 0 δεν καταχωρείται στο αρχείο, οπότε κατά το διάβασμα από αυτό το πρόγραμμα

δεν σταματάει, ακόμα και όταν φτάσω στο τέλος του αρχείου, το δε EOFδεν μπορώ να το χρησιμοποιήσω. Μια άλλη λύση εκτός από την χρήση της συνάρτησης feof( ) θα μπορούσε να είναι η παρακάτω (όσον αφορά το τμήμα του προγράμματος που αφορά το διάβασμα από το αρχείο):

```
int  k;
.....................................................
fptr = fopen ("C:/data/akeraioi.txt", "r");
if (fptr == NULL) {
     printf ("Error");
     exit (EXIT_FAILURE); }
while (1) {
     k = fscanf (fptr, "%d", &ak);
     if (k !=1)
          break;
     else
          printf ("%d", ak); }
fclose (fptr);
```

9.3.6. fputs( ). *Γράφει σε ένα αρχείο μια ομάδα χαρακτήρων.* Είναι παρόμοια με την puts( ). Δέχεται 2 ορίσματα:

- Το πρώτο είναι δείκτης σε χαρακτήρα, ο οποίος δείχνει στην πρώτη θέση της μνήμης όπου βρίσκονται οι χαρακτήρες που θα γραφούν στο αρχείο.
- Το δεύτερο όρισμα καθορίζει την ροή στην οποία θα σταλούν οι χαρακτήρες.

Επιστρέφει ένα αρνητικό αριθμό ή EOF εάν η εγγραφή δεν είναι επιτυχής. Η δήλωσή της είναι:

```
int  fputs (char*, FILE*);
```

Το παρακάτω πρόγραμμα διαβάζει μια συμβολοσειρά από το πληκτρολόγιο και την αποθηκεύει στο αρχείο points.txt. Δεν αποθηκεύει τον ειδικό χαρακτήρα '\x0':

```
FILE  *fptr;
char pin[30];
fptr = fopen ("C:/data/points.txt", "w");
gets(pin);
fputs(pin, fptr);
fclose (fptr);
```

9.3.7. fgets( ). *Διαβάζει από αρχείο μια ομάδα χαρακτήρων.* Είναι παρόμοια με την gets( ). Δέχεται 3 ορίσματα:

- Το πρώτο είναι δείκτης σε χαρακτήρα, ο οποίος δείχνει στην πρώτη θέση της μνήμης όπου θα αποθηκευτεί η ομάδα χαρακτήρων ως συμβολοσειρά (δηλαδή τίθεται και ο χαρακτήρας '\x0' στο τέλος) .
- Το δεύτερο όρισμα είναι ακέραιος, έστω ο num. Η συνάρτηση διαβάζει χαρακτήρες μέχρι να συναντήσει χαρακτήρα αλλαγής γραμμής ή num-1 χαρακτήρες, όποιο συμβεί πρώτο.
- Το τρίτο όρισμα καθορίζει την ροή από την οποία θα διαβαστούν οι χαρακτήρες.

Επιστρέφει ένα δείκτη σε χαρακτήρα όσο το πρώτο της όρισμα. Σε αδυναμία διαβάσματος επιστρέφει NULL. Η δήλωσή της είναι:

char* fgets (char *, int, FILE*);

Το επόμενο παράδειγμα αποτελεί παραλλαγή αυτού της παραγράφου 9.3.5. Εδώ η συνάρτηση fgets( ) διαβάζει ομάδες χαρακτήρων από το αρχείο points.txt μέχρι να φτάσουμε στο τέλος του. Τις ομάδες χαρακτήρων αυτές τις εμφανίζει ως συμβολοσειρές στην οθόνη:

```
FILE  *fptr;
char pin[30];

fptr = fopen ("C:/data/points.txt", "r");
while (!feof(fptr)) {
      fgets(pin, 10, fptr);
      puts(pin);}
fclose (fptr);
```

9.4. ΠΡΟΣΠΕΛΑΣΗ ΔΥΑΔΙΚΩΝ ΑΡΧΕΙΩΝ.

Συναρτήσεις που διέπουν τα δυαδικά αρχεία είναι οι:

9.4.1. fwrite( ). Χρησιμοποιείται κυρίως για την αποθήκευση ενός αριθμού από byte άμεσα από κάποια θέσης μνήμης σε ένα αρχείο. Δέχεται τέσσερα ορίσματα:

- Το πρώτο είναι δείκτης σε void, ο οποίος δείχνει στην πρώτη θέση της μνήμης όπου βρίσκονται τα byte που θα γραφούν στο αρχείο.
- Το δεύτερο είναι ακέραιος, ο οποίος καθορίζει το μέγεθος ενός στοιχείου που θα αποθηκευτεί, σε byte.
- Το τρίτο είναι ακέραιος, ο οποίος καθορίζει το πλήθος των στοιχείων που θα αποθηκευτούν.
- Το τέταρτο καθορίζει την ροή προς το αρχείο.

Η συνάρτηση επιστρέφει το πλήθος των στοιχείων που αποθηκεύονται στο αρχείο. Αν αυτή η τιμή επιστροφής δηλαδή είναι ίση με το τρίτο της όρισμα, τότε η fwrite( ) εκτελέστηκε επιτυχώς, αλλιώς απέτυχε. Η δήλωσή της είναι:

int fwrite (void*, int, int, FILE*);

Το γράψιμο ξεκινά από το σημείο που δείχνει ο δείκτης θέσης αρχείου, αφού γίνει δε η εγγραφή, ο δείκτης μετακινείται προς «τα δεξιά» (προς το τέλος του αρχείου δηλαδή) κατά τόσες θέσεις, όσα είναι και τα byte που έγραψε.

Στο παράδειγμα που ακολουθεί αποθηκεύονται σε ένα αρχείο με το όνομα binary οι ακέραιοι αριθμοί από το 1 έως το 10:

```
FILE  *fptr;
int  k;

fptr = fopen ("C:/data/binary", "wb");
for (k=1; k<=10; k++)
     fwrite (&k, sizeof(int), 1, fptr);
fclose (fptr);
```

Στο επόμενο παράδειγμα αποθηκεύονται σε ένα αρχείο με το όνομα binary1 οι κεφαλαίοι Λατινικοί χαρακτήρες:

```
FILE  *fptr;
char  ch;

fptr = fopen ("C:/data/binary1", "wb");
for (ch='A'; ch<='Z'; ch++)
     fwrite (&ch, sizeof(char), 1, fptr);
fclose (fptr);
```

9.4.2. fread( ). Χρησιμοποιείται για το διάβασμα από ένα αρχείο ενός πλήθους στοιχείων κάποιου μεγέθους (σε byte) και τα τοποθετεί σε κάποια περιοχή μνήμης. Δέχεται τέσσερα ορίσματα:

- Το πρώτο είναι δείκτης σε void, ο οποίος δείχνει στην πρώτη θέση της μνήμης όπου θα αποθηκευτούν τα byte που θα διαβαστούν από το αρχείο.
- Το δεύτερο είναι ακέραιος, ο οποίος καθορίζει το μέγεθος ενός στοιχείου που θα διαβαστεί, σε byte.
- Το τρίτο είναι ακέραιος, ο οποίος καθορίζει το πλήθος των στοιχείων που θα διαβαστούν.
- Το τέταρτο καθορίζει την ροή από το αρχείο.

Η συνάρτηση επιστρέφει το πλήθος των στοιχείων που διάβασε. Η δήλωσή της είναι:

int fread (void*, int, int, FILE*);

Το διάβασμα ξεκινά από το σημείο που δείχνει ο δείκτης στο αρχείο, αφού δε ολοκληρωθεί, ο δείκτης μετακινείται προς «τα δεξιά» (προς το τέλος του αρχείου δηλαδή) κατά τόσες θέσεις, όσα είναι και τα byte που διάβασε.

Το παράδειγμα που ακολουθεί είναι συμπληρωματικό του πρώτου της παραπάνω παραγράφου 9.4.1. Στο αρχείο binary είχαμε αποθηκεύσει τους ακέραιους από το 1 έως το 10. Εδώ διαβάζουμε τα περιεχόμενα αυτού του αρχείου και τα εμφανίζουμε στην οθόνη (οι ακέραιοι δηλαδή από 1 έως 10):

```
FILE  *fptr;
int pin[30], k;

fptr = fopen ("C:/data/binary", "rb");
fread (pin, sizeof(int), 10, fptr);
for (k=0; k<10; k++)
     printf("%5d", pin[k]);
fclose (fptr);
```

Το επόμενο παράδειγμα είναι συμπληρωματικό του δεύτερου της παραπάνω παραγράφου 9.4.1. Στο αρχείο binary1.txt είχαμε αποθηκεύσει τους κεφαλαίους Λατινικούς χαρακτήρες. Εδώ διαβάζουμε τα περιεχόμενα αυτού του αρχείου και τα εμφανίζουμε στην οθόνη:

```
FILE  *fptr;
char pin[30], k;

fptr = fopen ("C:/data/binary1", "rb");
fread (pin, sizeof(char), 'Z'-'A'+1, fptr);
for (k=0; k<='Z'-'A'; k++)
     printf("%5c", pin[k]);
fclose (fptr);
```

9.5. ΤΥΧΑΙΑ ΠΡΟΣΠΕΛΑΣΗ ΑΡΧΕΙΩΝ.

Όταν μιλούμε για **τυχαία προσπέλαση**, εννοούμε την δυνατότητα που έχουμε να μετακινήσουμε τον δείκτη θέσης αρχείου σε οποιαδήποτε θέση του. Το γράψιμο στο αρχείο και το διάβασμα από αυτό θα γίνει στην ή από την θέση στην οποία έχουμε μεταβεί. Συναρτήσεις τυχαίας προσπέλασης αρχείων είναι οι παρακάτω:

9.5.1.fseek( ). Δέχεται τρία ορίσματα:

- Το πρώτο καθορίζει την ροή προς το αρχείο.
- Το δεύτερο είναι μακρύς ακέραιος, ο οποίος καθορίζει την απόσταση σε byte από την θέση που καθορίζει το τρίτο όρισμα.
- Το τρίτο είναι ακέραιος, ο οποίος μπορεί νά έχει τιμή 0, 1 ή 2. Η τιμή 0 (η οποία λέγεται και SEEK_SET) αντιστοιχεί στην αρχή του αρχείου, η 1(η οποία λέγεται και SEEK_CUR) αντιστοιχεί στην τρέχουσα θέση και η 2 (η οποία λέγεται και SEEK_END) αντιστοιχεί στο τέλος του αρχείου.

Η συνάρτηση επιστρέφει 0 εάν εκτελεστεί με επιτυχία, αλλιώς επιστρέφει μη μηδενική τιμή. Η δήλωσή της είναι:

int fseek (FILE*, long, int);

Στο παρακάτω πρόγραμμα ανοίγουμε για διάβασμα ένα αρχείο κειμένου, το points.txt. Τοποθετούμε τον δείκτη του αρχείου 5 θέσεις πριν το τέλος του και διαβάζουμε τον χαρακτήρα που βρίσκεται εκεί. Στη συνέχεια τον εμφανίζουμε στην οθόνη:

```
FILE  *fptr;
int  k;

fptr = fopen ("C:/data/points.txt", "r");
fseek (fptr,-5,2 );
k = fgetc(fptr);
putchar(k);
fclose (fptr);
```

9.5.2. rewind( ). Τοποθετεί τον δείκτη θέσης στην αρχή του αρχείου. Το όρισμά της καθορίζει την ροή προς το αρχείο. Η δήλωσή της είναι:

int rewind (FILE*);

9.5.3. ftell( ). Επιστρέφει την τρέχουσα τιμή του δείκτη θέσης ενός αρχείου. Το όρισμά της καθορίζει την ροή προς το αρχείο. Η δήλωσή της είναι:

longftell (FILE*);

Εάν έχει συμβεί λάθος, επιστρέφει τιμή -1.

Παράδειγμα 1.

Το παρακάτω πρόγραμμα:

```
FILE  *fptr;
char  ch;
int  gch;
long  lak;

fptr = fopen ("C:/data/binary", "wb");
lak = ftell (fptr);
printf ("%ld\n", lak);

for (k=1; k<=10; k++) {
      fwrite (&k, sizeof(int), 1, fptr);
      lak = ftell(fptr);
      printf ("%ld\n", lak); }

fclose (fptr);
```

θα εμφανίσει στην οθόνη:

0
4
8
12
16
20
24
28
32
36
40

Παράδειγμα 2.

Στο πρόγραμμα που ακολουθεί υποθέτουμε ότι δίνουμε από το πληκτρολόγιο τους χαρακτήρες qwerty$

```
FILE  *fptr;
int  k;
long  lak;
char ch, gch;

fptr = fopen ("C:/data/points.txt", "w");
lak = ftell (fptr);
printf ("%ld\n", lak);
if (fptr == NULL)
     exit (1);

ch = getche( );
while (ch != '$') {
     fputc (ch, fptr);
     lak = ftell (fptr);
     printf ("%ld\n", lak);
     ch = getche( ); }

fclose (fptr);

fptr = fopen ("C:/data/points.txt", "r");
if (fptr == NULL)
     exit (1);

while (!feof (fptr)) {
     lak = ftell (fptr);
     printf ("%ld\n", lak);
     gch = fgetc (fptr);
     putchar (gch); }

fclose (fptr);
```

Στην οθόνη θα εμφανιστεί:

0
q1
w2
e3

r4
t5
y6
$0
q1
w2
e3
r4
t5
y6

Παράδειγμα 3.

Στο παρακάτω πρόγραμμα ζητούνται τα στοιχεία ενός πίνακα ακεραίων Ν θέσεων, τα οποία αποθηκεύονται σε ένα δυαδικό αρχείο, το pinax.bin. Στη συνέχεια ζητείται ένας ακέραιος, ο num και το πρόγραμμά ανακαλεί από το αρχείο τον num ακέραιο του πίνακα:

```
FILE *fptr=NULL;
int  pin[N], num;
int  k, ak;

for (k=0; k<N; k++)
        scanf (“%d”, &pin[k]);

if ((fptr = fopen("C:/pinax.bin", "wb")) != NULL) {

        fwrite (pin, sizeof(pin), 1, fptr);

        fclose(fptr);

        if ((fptr =fopen("C:/pinax.bin", "rb")) == NULL) {
                printf ("Error: %s\n", strerror(errno));
                exit (1); }

        printf ("Position ? ");
        scanf("%d",&num);

        if (fseek(fptr,(num-1)*sizeof(int), SEEK_SET)){
                printf("Out of orders\n");
                exit(1);}

        fread(&ak, sizeof(int), 1 , fptr);

        printf ("Element %d is %d", num, ak);}

else {
        printf ("Error: %s\n", strerror(errno));
        exit (1); }

fclose(fp);
```

Παράδειγμα 3.

Ένα ακόμη παράδειγμα γραφής από αρχείο σε αρχείο:

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

```
FILE  *in, *out;
char ch;

if (argc!=3) {
     printf ("LEIPOYN ONOMATA ARXEIWN\n");
     exit( 1); }
if ((in = fopen(argv[1], "rb"))==NULL){
     printf ("DEN YPARXEI ARXEIO PROELEYSHS\n");
     exit(1); }
if ((out = fopen(argv[2], "wb"))==NULL){
     printf ("DEN YPARXEI ARXEIO PROORISMOY\n");
     exit (1); }
while (!feof(in)){
     ch = getc (in);
     putc (ch, out);}
fclose (in);
fclose (out); }
```

9.6. ΔΙΑΓΡΑΦΗ ΑΡΧΕΙΩΝ – ΧΕΙΡΙΣΜΟΣ ΛΑΘΩΝ.

Η συνάρτηση **remove( )** σβήνει ένα αρχείο. Η δήλωσή της είναι η εξής:

```
int  remove (char  *);
```

Στο παρακάτω παράδειγμα, μετά το διάβασμα από το αρχείο points.txt, το αρχείο αυτό διαγράφεται:

```
FILE  *fptr;
char pin[30];

fptr = fopen ("C:/data/points.txt", "r");
while (!feof(fptr)) {
     fgets (pin, 10, fptr);
     puts (pin);}
fclose (fptr);
remove ("C:/data/points.txt");
```

Η συνάρτηση **ferror( )** επιστρέφει τιμή αληθή εάν κατά την τελευταία λειτουργία που εφαρμόστηκε σε ένα αρχείο συνέβη λάθος, αλλιώς επιστρέφει τιμή ψευδή. Η δήλωσή της είναι η εξής:

```
int  ferror (FILE  *);
```

9.7. ΠΡΟΓΡΑΜΜΑΤΑ ΜΕ ΠΟΛΛΑ ΑΡΧΕΙΑ.

Στον κλασικό μικρής κλίμακας προγραμματισμό, το πρόγραμμα που γράφουμε αποτελείται από ένα μόνο αρχείο. Στην πράξη, ένα πρόγραμμα στην C μπορεί να

αποτελείται από περισσότερα από ένα αρχεία. Η πρακτική του χωρισμού σε πολλά αρχεία είναι ιδιαίτερα χρήσιμη, αφού βοηθάει τόσο στην καλύτερη δομή του προγράμματος, όσο και στον έλεγχο και την διόρθωση λαθών.

Θα δείξουμε την τεχνική μεταγλώττισης προγράμματος με πολλά αρχεία χρησιμοποιώντας το περιβάλλον της DevC++. Στο παράδειγμα που ακολουθεί, το πρόγραμμά μας περιέχει τρεις συναρτήσεις, τις message( ), display( ) και final( ):

```
#include <stdio.h>

void message( );
void display(char *);
void final(char *);

int main(void) {
    message( );
    display("ARISTA");
    final("ERGASIAS\n");
    return 0; }

void message( ) {
    printf ("EPITYXIA!!!\n"); }

void display(char *grade) {
    printf ("BA8MOLOGEISTE ME %s!\n", grade); }

void final(char *msg) {
    printf("TELOS %s\n", msg); }
```

Στο περιβάλλον της DEV C++ δημιουργούμε ένα Emptyproject για C με όνομα για παράδειγμα test. Δημιουργούμε τρία χωριστά αρχεία τα οποία προσθέτουμε στο test, χρησιμοποιώντας την επιλογή AddtoProject. Τα τρία αυτά αρχεία είναι τα εξής:

α) Το αρχείο που θα περιέχει τις δηλώσεις των συναρτήσεων που θα χρησιμοποιήσουμε:

```
void  message( );
void  display(char *);
void  final(char *);
```

Το δημιουργούμε ως αρχείο κεφαλίδας (header, με προέκταση .h). Το ονομάζουμε για παράδειγμα **library.h** και θα βρίσκεται αποθηκευμένο στον τρέχοντα κατάλογο του προγράμματός μας.

β) Το αρχείο που θα περιέχει τους ορισμούς των συναρτήσεών μας:

```
void message( ) {
   printf ("EPITYXIA!!!\n"); }

void display(char *grade) {
   printf ("BA8MOLOGEISTE ME %s!\n", grade); }

void final(char *msg) {
   printf ("TELOS %s\n", msg); }
```

Συνηθίζουμε να του δίνουμε το ίδιο κύριο όνομα με το αρχείο κεφαλίδας, αλλά με προέκταση .c. Έτσι, το ονομάζουμε για παράδειγμα **library.c**.

γ) Το αρχείο που θα περιέχει την συνάρτηση main( ), (το ονομάζουμε π.χ. **main.c**):

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

#include "library.h"

int main (void) {
    message( );
    display ("ARISTA");
    final ("ERGASIAS\n");
    return 0; }
```

Παρατηρείστε ότι το δικό μας αρχείο βιβλιοθήκης library.h βρίσκεται μέσα σε " " και όχι μέσα σε <>, αναγκάζοντας έτσι τον compiler να το αναζητήσει μέσα στον ίδιο κατάλογο με αυτόν όπου βρίσκεται το αρχείο main.

9.8. ΕΞΩΤΕΡΙΚΕΣ ΜΕΤΑΒΛΗΤΕΣ ΣΕ ΠΡΟΓΡΑΜΜΑΤΑ ΜΕ ΠΟΛΛΑ ΑΡΧΕΙΑ.

Μια ερώτηση που προκύπτει είναι η εξής: εάν σε ένα πρόγραμμα με πολλά αρχεία δηλώσουμε μια εξωτερική μεταβλητή, αυτή είναι γνωστή και στα υπόλοιπα αρχεία; Ας δούμε μια παραλλαγή του προγράμματος της παραγράφου 7.2.:

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

void message( );
void display(char *);
void final(char *);

int ak;
int main(void) {
      scanf ("%d", &ak);
      message( );
      display("PHRATE ");
      final("ERGASIAS\n");
      return 0; }

void message( ) {
      if (ak >= 5)
            printf ("EPITYXIA!!!\n");
      else
            printf ("APOTYXATE\n"); }

void display(char *grade) {
      printf ("%s %d\n", grade, ak); }

void final(char *msg) {
      printf("TELOS %s\n", msg); }
```

Αν θελήσουμε τώρα να το «διασπάσουμε» σε πολλά αρχεία, θα έχουμε:

α) Για το αρχείο **library.h** ό,τι και προηγούμενο παράδειγμα, δηλαδή:

```
void message( );
void display(char *);
void final(char *);
```

β) Για το αρχείο **library.c**:

```
void message( ) {
   extern int ak;
   if (ak >= 5)
      printf ("EPITYXIA!!!\n");
   else
      printf ("APOTYXATE\n"); }

void display(char *grade) {
   extern int ak;
   printf ("%s %d\n", grade, ak); }

void final(char *msg) {
   printf("TELOS %s\n", msg); }
```

γ) Για το αρχείο **main.c**:

```
#include <stdio.h>

#include "library.h"

int ak;

int main(void) {
   scanf ("%d", &ak);
   message ( );
   display ("PHRATE");
   final ("ERGASIAS\n");
   return 0; }
```

Η δήλωση extern είναι απαραίτητη στις συναρτήσεις του library.c, αφού η ak είναι μεταβλητή δηλωμένη σε ένα άλλο αρχείο.

ΚΕΦΑΛΑΙΟ 10

ΒΙΒΛΙΟΘΗΚΕΣ ΤΗΣ C

Αναφερόμαστε σε κάποιες από τις τυποποιημένες βιβλιοθήκες στης C και σε συναρτήσεις που περιλαμβάνουν και οι οποίες δεν έχουν αναφερθεί ή αναλυθεί μέχρι τώρα:

10.1. Η ΒΙΒΛΙΟΘΗΚΗ string.h

10.1.1. Η συνάρτηση **memcpy( )**. Χρησιμοποιείται για την αντιγραφή οποιουδήποτε τύπου δεδομένων από μια περιοχή μνήμης σε μια άλλη. Η δήλωσή της είναι:

void *memcpy (void *dest, void *src, size_t num);

Αντιγράφει num byte από τη θέση μνήμης στην οποία δείχνει ο δείκτης src στην θέση μνήμης και κάτω στην οποία δείχνει ο δείκτης dest. Οι περιοχές μνήμης δεν πρέπει να επικαλύπτονται.

Στο παρακάτω παράδειγμα δεσμεύουμε χώρο για ak ακέραιους, τους οποίους στη συνέχεια δίνουμε από το πλήκτρολόγιο. Κατόπιν δεσμεύουμε χώρο για num ακεραίους, αντιγράφουμε εκεί τις τιμές που διαβάστηκαν προηγουμένως και εμφανίζουμε τα περιεχόμενα του δεύτερου χώρου που δεσμεύσαμε:

```
int *ptr, *dkt, k, ak, num;

scanf("%d", &ak);
ptr = (int *) malloc(ak*sizeof(int));
for (k=0; k<ak; k++)
     scanf("%d", ptr+k);
scanf("%d", &num);
dkt = (int *) malloc(num*sizeof(int));
memcpy(dkt, ptr, ak*sizeof(int));
for (k=0; k<num; k++)
     printf("%d\n", *(dkt+k));
```

10.1.2. Η συνάρτηση **memmove( )**. Είναι παρόμοια με την memcpy( ). Δουλεύει σωστά, ακόμη και εάν οι περιοχές μνήμης επικαλύπτονται. Πρέπει να έχει δεσμευτεί αρκετός χώρος στον προορισμό, ώστε να εξασφαλιστεί η μη μεταφορά δεδομένων σε μη δεσμευμένο χώρο μνήμης.

10.1.3. Η συνάρτηση **memcmp( )**. Η δήλωσή της είναι:

int memcmp (void *ptr, void *dkt, size_t num);

Συγκρίνει num byte στις περιοχές οι οποίες δείχνονται από τους δείκτες ptr και dkt. Εάν είναι ίδια επιστρέφει 0 αλλιώς επιστρέφει <0 ή >0, όπως και η

strcmp( ). Στο παρακάτω παράδειγμα συγκρίνονται ακέραιοι, οι οποίοι τοποθετήθηκαν σε δυναμικά δεσμευμένες περιοχές μνήμης:

```
int *ptr, *dkt, k, ak, num, x;

scanf("%d", &ak);
ptr = (int *) malloc(ak*sizeof(int));

for (k=0; k<ak; k++)
     scanf("%d", ptr+k);

scanf("%d", &num);
dkt = (int *) malloc(num*sizeof(int));

for (k=0; k<num; k++)
     scanf("%d", dkt+k);

x = memcmp(ptr, dkt, ak*sizeof(int));

printf("%d\n", x);
```

Αν δώσετε για παράδειγμα τιμή στο ak 3 και τιμές 1, 2και 3 και στο num επίσης 3 και τιμές 2, 3 και 4, θα πάρουμε αποτέλεσμα -1. Για τιμές 2, 4 και 5 και 1, 3 και 6 θα πάρουμε αποτέλεσμα 1.

10.1.4. Η συνάρτηση **memset( )**. Η δήλωσή της είναι:

```
void  *memset (void *ptr, char ch, size_t num);
```

Κάνει την τιμή των numbyte της μνήμης, στην οποία δείχνει ο δείκτης ptr ίση με ch.

Το παρακάτω πρόγραμμα δεσμεύει χώρο για ak χαρακτήρες και θέτει σε όλο τον χώρο που δέσμευσε και θέτει σε όλες τις θέσεις τον χαρακτήρα B:

```
char *ptr;
int k, ak;

scanf("%d", &ak);
ptr = (char *) malloc(ak));

memset(ptr, 'B', ak);

for (k=0; k<ak; k++)
     printf("%c\n", *(ptr+k));
```

Τροποποιώντας το παραπάνω πρόγραμμα δεσμεύουμε χώρο για ak ακέραιους. Στο κάθε byte του κάθε ακεραίου τοποθετούμε τον χαρακτήρα A και κατόπιν τους εμφανίζουμε στην οθόνη. Προφανώς στην οθόνη θα γραφεί (σε μια στήλη) το A, 4*ak φορές:

```
scanf("%d", &ak);
ptr = (char *) malloc(ak*sizeof(int));

memset(ptr, 'A', ak*sizeof(int));

for (k=0; k<ak*sizeof(int); k++)
     printf("%c\n", *(ptr+k));
```

10.1.5. Η συνάρτηση **strcspn( )**. Η δήλωσή της είναι:

int strcspn (char *str, char *mat);

Επιστρέφει την θέση του πρώτου χαρακτήρα της συμβολοσειράς str, ο οποίος υπάρχει και στην συμβολοσειρά mat. Στο παρακάτω πρόγραμμα δηλαδή, αν δώσουμε ως συμβολοσειρές τις giannis και maria, στην οθόνη θα εμφανίσει 1.

```
char str[N], mat[N];
.....................
scanf("%s", str);
scanf("%s", mat);
printf("%d\n", strcspn(str, mat));
```

10.1.6. Η συνάρτηση **strncat( )**. Η δήλωσή της είναι:

char *strncat (char *pin, char *mat, int num);

Παρόμοια με την strcat( ), όμως επικολλά στο τέλος της συμβολοσειράς pin τους num πρώτους χαρακτήρες της συμβολοσειράς mat και προσθέτει στο τέλος τον χαρακτήρα \x0.

10.1.7. Η συνάρτηση **strncpy( )**. Η δήλωσή της είναι:

char *strncpy (char *pin, char *mat, int num);

Παρόμοια με την strcpy( ), όμως αντιγράφει τους num πρώτους χαρακτήρες της συμβολοσειράς mat στην θέση όπου δείχνει ο δείκτης pin.

10.1.8. Η συνάρτηση **strerror( )**. Δέχεται ως όρισμα ένα ακέραιο, τον αριθμό λάθους και επιστρέφει ένα δείκτη σε χαρακτήρα, ο οποίος δείχνει στην αρχή μιας συμβολοσειράς όπου υπάρχει το μήνυμα λάθους.

10.1.9. Η συνάρτηση **strchr( )**. Η δήλωσή της είναι:

char * strchr (char *str, intnum);

Αναζητάει την πρώτη εμφάνιση του χαρακτήρα που αντιστοιχεί στο num, στην συμβολοσειρά str. Επιστρέφει ένα δείκτη στη θέση που βρήκε τον χαρακτήρα, αλλιώς NULL.

10.1.10. Η συνάρτηση **strrchr( )**. Η δήλωσή της είναι:

char * strrchr (char *str, intnum);

Παρόμοια με την strchr( ), όμως αναζητά την τελευταία εμφάνιση του χαρακτήρα που αντιστοιχεί στο num, στην συμβολοσειρά str. Επιστρέφει ένα δείκτη στη θέση που βρήκε τον χαρακτήρα, αλλιώς NULL.

10.1.11. Η συνάρτηση **strstr( )**. Η δήλωσή της είναι:

char * strstr (char *pin,char *mat);

Αναζητάει την πρώτη εμφάνιση της συμβολοσειράς στην οποία δείχνει ο mat μέσα στην συμβολοσειρά στην οποία δείχνει ο pin. Επιστρέφει ένα δείκτη στη θέση που βρήκε την συμβολοσειρά αυτή, αλλιώς NULL.

10.1.12. Η συνάρτηση **strtok( )**. Η δήλωσή της είναι:

char * strtok (char *str, char *set);

Διαχωρίζει τη συμβολοσειρά στην οποία δείχνει ο str σύμφωνα με τους χαρακτήρες της συμβολοσειράς στην οποία δείχνει ο set. Η συνάρτηση καλείται συνεχώς και κάθε φορά επιστρέφει το νέο τμήμα που επισημαίνει. Αυτό γίνεται μέχρι το τέλος της κύριας συμβολοσειράς (της str δηλαδή). Η συμβολοσειρά την οποία χωρίζουμε δίνεται ως όρισμα μόνο κατά την πρώτη κλήση, ενώ κατά τις υπόλοιπες το όρισμα τίθεται ίσο με NULL. Αυτός είναι ένας τρόπος για να ειδοποιήσετε την strtok( ) κατά πόσον ξεκινάτε μια καινούργια διαδικασία διαχωρισμού ή απλώς συνεχίζετε τον διαχωρισμό από προηγουμένως. Η strtok( ) κατά κάποιον τρόπο «θυμάται» την προηγούμενη κατάσταση και γι' αυτό δεν είναι ασφαλής σε εφαρμογές πολυνηματικού προγραμματισμού. Ακόμη, τροποποιεί την αρχική συμβολοσειρά, θέτοντας τον χαρακτήρα \x0 στις θέσεις που υπάρχουν χαρακτήρες της set.

Το παρακάτω πρόγραμμα:

```
char str[80] = "This is a matrix string";
char set[ ] = "u";
char *token;

token = strtok(str, set);

while( token != NULL ) {
      printf( "%s\n", token );
      token = strtok(NULL, set); }
```

θα εμφανίσει στην οθόνη:

```
This
is
a
matrix
string
```

εάν δε δώσετε μετά την εντολή **printf("%s", str);** θα εμφανίσει:

```
This
```

Το ίδιο αποτέλεσμα θα είχαμε εάν οι str και set είχαν δηλωθεί ως εξής:

```
char str[80] = "This, is a matrix-string";
charset[ ] = " , -";
```

ενώ εάν ο set ήταν ο παρακάτω:

```
charset[ ] = ",-";
```

στην οθόνη θα παίρναμε:

```
This
uisamatrix
string
```

Το παρακάτω πρόγραμμα

```
char str[80] = "This, is a matrix-string";
char set[ ] = "-,u";
char *token;
FILE *fptr;

fptr = fopen("C:/data/tok.txt", "w");
if (fptr != NULL) {
      token = strtok (str, set);
      while(token != NULL ) {
            fputs(token, fptr);
            token = strtok(NULL, s);}}
```

αποθηκεύει στο αρχείοto k.txt τηνφράση:

```
Thisisamatrixstring
```

10.2. Η ΒΙΒΛΙΟΘΗΚΗ time.h

10.2.1. Η συνάρτηση **time( )**. Η δήλωσή της είναι:

```
time_t  time (time_t *);
```

Επιστρέφει τον χρόνο από τις 00:00 της 1/1/1970 σε δευτερόλεπτα.Μπορείτε να την καλέσετε με ένα από τους εξής δύο τρόπους:

```
time_t seconds;
seconds = time(NULL);
```

ή

```
time(&seconds);
```

10.2.2. Η συνάρτηση **clock( )**. Η δήλωσή της είναι:

```
clock_tclock (void);
```

Το clock_t είναι στην πραγματικότητα ένας ακέραιος (long) τύπος. Η συνάρτηση επιστρέφει τον χρόνο που έχει περάσει από την έναρξη εκτέλεσης του προγράμματος σε παλμούς χρονισμού του επεξεργαστή. Αν θέλουμε να μετρήσουμε τον χρόνο σε δευτερόλεπτα, πρέπει να διαιρέσουμε την τιμή αυτή με την σταθερά CLOCKS_PER_SEC.

Μπορούμε να μετρήσουμε τον χρόνο που έχει μεσολαβήσει μεταξύ δύο σημείων του προγράμματος καλώντας δυο φορές την συνάρτηση, μια πριν και μια μετά το μπλοκ εντολών που μας ενδιαφέρει και παίρνοντας την διαφορά των δύο τιμών, όπως στο παράδειγμα που ακολουθεί:

```
clock_t  arxi, telos;
double  elaps;
int i;

arxi = clock();
printf ("A big loop begins at arxi = %ld\n", arxi);
for(i=0; i< 10000000; i++)
    ;
telos = clock();
printf ("The big loop ends at telos = %ld\n", telos);
elaps = (double) (telos - arxi) / CLOCKS_PER_SEC;
printf ("Total CPU time: %lf\n", elaps);
```

Στην οθόνη θα εμφανιστεί κάτι τέτοιο:

```
A big loop begins at arxi = 15
The big loop ends at telos = 46
Total CPU time = 0.031000
```

10.2.3. Η συνάρτηση **difftime( )**. Η δήλωσή της είναι:

```
double difftime (time_t  t1,  time_t  t2);
```

Επιστρέφει την χρονική διαφορά των t1 και t2 σε δευτερόλεπτα.

10.2.4. Η συνάρτηση **localtime( )**. Η δήλωσή της είναι:

```
stuct  tm *localtime (time_t  *ptr);
```

Μετατρέπει τη χρονική πληροφορία που περιέχεται στην μεταβλητή στην οποία δείχνει ο ptr σε μια δομή του τύπου tm και επιστρέφει δείκτη σε αυτή τη δομή. Ηδομήtmπεριγράφεταιωςεξής:

```
struct tm {
   int tm_sec;   /* seconds,  range 0 to 59  */
   int tm_min;   /* minutes, range 0 to 59        */
   int tm_hour;  /* hours, range 0 to 23          */
   int tm_mday;  /* day of the month, range 1 to 31 */
   int tm_mon;   /* month, range 0 to 11           */
   int tm_year;  /* The number of years since 1900  */
   int tm_wday;  /* day of the week, range 0 to 6   */
   int tm_yday;  /* day in the year, range 0 to 365 */
   int tm_isdst; /* daylight saving time           */
};
```

10.2.5. Η συνάρτηση **asctime( )**. Η δήλωσή της είναι:

char *asctime (struct tm*ptr);

Μετατρέπει τη χρονική πληροφορία που περιέχεται στην δομή στην οποία δείχνει ο δείκτηςptr σε συμβολοσειρά που έχει τη μορφή:

"Wed Jan 15 11:30:25 2020"

Το παρακάτω πρόγραμμα θα εμφανίσει μια τέτοια συμβολοσειρά και επιπλέον τους αριθμούς 14 και 0 (για την θερινή ώρα).

```
time_t wra;
struct tm *ptr;

time( &wra);
ptr = localtime(&wra);
printf ("Current local time and date: %s\n", asctime(ptr));
printf ("%d\n", ptr->tm_yday);
printf ("%d\n", ptr->tm_isdst);
```

10.3. Η ΒΙΒΛΙΟΘΗΚΗ limits.h

Περιέχει ένα πλήθος από σταθερές ακεραίου τύπου, οι οποίες είναι ορισμένες με μορφή μακροεντολών. Έτσι:

1. **CHAR_BIT**: Η μέγιστη τιμή του αριθμού των bit για την αναπαράσταση ενός χαρακτήρα. Τυπική τιμή το 8.
2. **SCHAR_MIN** : Η ελάχιστη τιμή για προσημασμένο χαρακτήρα. (-128)
3. **SCHAR_MAX** : Η μέγιστη τιμή για προσημασμένο χαρακτήρα. (+128)
4. **UCHAR_MAX**: Η μέγιστη τιμή για μη προσημασμένο χαρακτήρα. (255)
5. **CHAR_MAX** : Η μέγιστη τιμή για χαρακτήρα. Τυπική τιμή 127.
6. **CHAR_MIN** : Η ελάχιστη τιμή για χαρακτήρα. Τυπική τιμή -128.
7. **MB_LEN_MAX** : Το μέγιστο πλήθος χαρακτήρων, οι οποίοι αποτελούν ένα χαρακτήρα πολλών byte. Τυπική τιμή 4.
8. **SHRT_MAX**: Η μέγιστη τιμή για «μικρό» (short) ακέραιο. Τυπική τιμή 32767.
9. **SHRT_MIN** :Η ελάχιστη τιμή για «μικρό» (short) ακέραιο. Τυπική τιμή -32768.
10. **USHRT_MAX** :Η μέγιστη τιμή για μη προσημασμένο «μικρό» (short) ακέραιο. Τυπική τιμή 65535.
11. **INT_MAX**: Η μέγιστη τιμή για ακέραιο. Τυπική τιμή 2147483647. Ταυτίζεται σε πολλές περιπτώσεις με το LONG_MAX.
12. **INT_MIN**: Η ελάχιστη τιμή για ακέραιο. Τυπική τιμή-2147483648. Ταυτίζεται σε πολλές περιπτώσεις με το LONG_MIN.

13. **UINT_MAX** : Η μέγιστη τιμή για μη προσημασμένο ακέραιο. Τυπική τιμή 65535. (Ταυτίζεται σε πολλές περιπτώσεις με το LONG_MAX)
14. **LONG_MAX**:Η μέγιστη τιμή για ακέραιο (ή μακρύ ακέραιο). Τυπική τιμή 2147483647.
15. **LONG_MIN**: Η ελάχιστη τιμή για ακέραιο (ή μακρύ ακέραιο). Τυπική τιμή -2147483648.
16. **ULONG_MAX**: Η μέγιστη τιμή για μη προσημασμένο ακέραιο. Τυπική τιμή4294967295.
17. **LLONG_MAX** : Η μέγιστη τιμή για πολύ μακρύ ακέραιο (με προσδιοριστή %lld) 9223372036854775807.
18. **LLONG_MIN** : Η ελάχιστη τιμή για πολύ μακρύ ακέραιο (με προσδιοριστή %lld) -9223372036854775808.

10.4. Η ΒΙΒΛΙΟΘΗΚΗ ctype.h

10.4.1. Η συνάρτηση **isalnum( )**. Η δήλωσή της είναι:

```
int isalnum(int);
```

Ελέγχει εάν το όρισμά της είναι αλφαριθμητικός χαρακτήρας.

10.4.2. Η συνάρτηση **isalpha( )**. Η δήλωσή της είναι:

```
int isalpha (int);
```

Ελέγχει εάν το όρισμά της είναι αλφαβητικός χαρακτήρας.

10.4.3. Η συνάρτηση **iscntrl( )**. Η δήλωσή της είναι:

```
int iscntrl (int);
```

Ελέγχει εάν το όρισμά της είναι χαρακτήραςελέγχου, δηλαδή κάποιος από τους:

\a	Ηχητικό σήμα
\b	Διάστημα πίσω
\n	Νέα γραμμή
\r	Επιστροφή στην αρχή της γραμμής
\t	Οριζόντιο προκαθορισμένο διάστημα (TAB)
\'	Εμφάνιση του απλού εισαγωγικού
\"	Εμφάνιση του διπλού εισαγωγικού
\\	Εμφάνιση της ανάποδης πλαγίας καθέτου

10.4.4. Η συνάρτηση **isdigit( )**. Η δήλωσή της είναι:

int isdigit (int);

Ελέγχει εάν το όρισμά της είναι ψηφίο του δεκαδικού αριθμητικού συστήματος.

10.4.5. Η συνάρτηση **isgraph( )**. Η δήλωσή της είναι:

int isgraph (int);

Ελέγχει εάν για το όρισμά της υπάρχει γραφική αναπαράστση. Αυτοί είναι όλοι οι χαρακτήρες εκτός από τους «λευκούς» χαρακτήρες (π.χ. διάστημα, TAB, αλλαγή γραμμής κλπ).

10.4.6. Η συνάρτηση **islower( )**. Η δήλωσή της είναι:

int islower (int);

Ελέγχει εάν το όρισμά της είναι πεζός (μικρός) λατινικός χαρακτήρας.

10.4.7. Η συνάρτηση **isupper( )**. Η δήλωσή της είναι:

int isupper (int);

Ελέγχει εάν το όρισμά της είναι κεφαλαίος λατινικός χαρακτήρας.

10.4.8. Η συνάρτηση **isprint( )**. Η δήλωσή της είναι:

int isprint (int);

Ελέγχει εάν το όρισμά της είναι εκτυπώσιμος χαρακτήρας.

10.4.9. Η συνάρτηση **ispunct( )**. Η δήλωσή της είναι:

int ispunct (int);

Ελέγχει εάν το όρισμά της είναι σημείο στίξης.

10.4.10. Η συνάρτηση **isspace( )**. Η δήλωσή της είναι:

int ispunct (int);

Ελέγχει εάν το όρισμά της είναι το κενό διάστημα.

10.4.11. Η συνάρτηση **isxdigit( )**. Η δήλωσή της είναι:

int isxdigit (int);

Ελέγχει εάν το όρισμά της είναι δεκαεξαδικό ψηφίο.

10.4.12. Η συνάρτηση **tolower( )**. Η δήλωσή της είναι:

int tolower(int);

Μετατρέπει τους κεφαλαίους χαρακτήρες σε πεζούς (μικρούς).Αν δοθεί άλλος χαρακτήρας, επιστρέφει τον χαρακτήρα αυτόν.

10.4.13. Η συνάρτηση **toupper( )**. Η δήλωσή της είναι:

int toupper(int);

Μετατρέπει τους πεζούς (μικρούς) χαρακτήρες σε κεφαλαίους.Αν δοθεί άλλος χαρακτήρας, επιστρέφει τον χαρακτήρα αυτόν.

10.5. Η ΒΙΒΛΙΟΘΗΚΗ math.h

Στις τριγωνομετρικές συναρτήσεις το όρισμα δίνεται σε ακτίνια (rad).

1. **sin( )** : Ημίτονο.
2. **asin( )** : Τόξο ημιτόνου.
3. **sinh( )** : Υπερβολικό ημίτονο.
4. **asinh( )** : Τόξο υπερβολικού ημιτόνου.
5. **cos( )** : Συνημίτονο.
6. **acos( )** : Τόξο συνημιτόνου.
7. **cosh( )** : Υπερβολικό συνημίτονο.
8. **acosh( )** : Τόξο υπερβολικού συνημιτόνου.
9. **tan( )** : Εφαπτομένη.
10. **atan( )** : Τόξο εφαπτομένης.
11. **tanh( )**: Υπερβολική εφαπτομένη.
12. **atanh( )** : Τόξο υπερβολικής εφαπτομένης τού ορίσματός της.
13. **atan2( )** : Δέχεται δύο παραμέτρους x και y. Υπολογίζει το τόξο υπερβολικής εφαπτομένης του x/y.
14. **abs( )** : Απόλυτη τιμή ακεραίου.
15. **fabs( )** : Απόλυτη τιμή.
16. **log( )** : Φυσικός λογάριθμος.
17. **log10( )** : Δεκαδικός λογάριθμος.
18. **exp( )** : Εκθετικό. Το e υψωμένο στο όρισμα.
19. **pow( )** : Δύναμη αριθμού.
20. **sqrt( )** : Τετραγωνική ρίζα αριθμού.
21. **cbrt( )** : Κυβική ρίζα αριθμού.
22. **ceil( )** : Υπολογίζει τον μικρότερο ακέραιο που είναι μεγαλύτερος από το όρισμά της.

23. **floor( )** : Υπολογίζει τον μεγαλύτερο ακέραιο που είναι μικρότερος από το όρισμά της
24. **hypot( )** : Δέχεται δύο παραμέτρους x και y. Υπολογίζει την υποτείνουσα του ορθογωνίου τριγώνου με κάθετες πλευρές τα x και y.

10.6. Η ΒΙΒΛΙΟΘΗΚΗ conio.h

Πρακτικά η βιβλιοθήκη αυτή χρησιμοποιείται για να εξασφαλίζει είσοδο-έξοδο για MS-DOS compilers. Χρησιμοποιούνται κατά βάση για τις συναρτήσεις **getch( )**και **getche( )**. Υπάρχει επίσης η συνάρτηση **putch( )** για την εμφάνιση ενός χαρακτήρα στην οθόνη (αντίστοιχη της putchar( )).

10.7. Η ΒΙΒΛΙΟΘΗΚΗ complex.h

Έχει ενταχθεί στο πρότυπο της C μετά την έκδοση C99. Αφορά τις λειτουργίες με μιγαδικούς αριθμούς.

10.8. Η ΒΙΒΛΙΟΘΗΚΗ float.h

Περιέχει ένα αριθμό από σταθερές που έχουν σχέση με float τιμές. Οι χρησιμότερες και πιο γνωστές είναι:

1. **FLT_MIN :** ελάχιστος float (αποθήκευση σε 4 byte). Τυπική τιμή 1.1754943508e-038
2. **DBL_MIN :** ελάχιστος double (αποθήκευση σε 8 byte). Τυπική τιμή 2.2250738585e-308
3. **LDBL_MIN :** ελάχιστος long double (αποθήκευση σε 16 byte). Τυπική τιμή 3.2052844739e-317
4. **FLT_MANT_DIG :** αριθμός ψηφίων για την βάση (mantissa) ενός float. Τυπική τιμή 24.
5. **DBL_MANT_DIG:** αριθμός ψηφίων για την βάση (mantissa) ενός double. Τυπική τιμή 53.
6. **LDBL_MANT_DIG:** αριθμός ψηφίων για την βάση (mantissa) ενός longdouble. Τυπική τιμή 64.
7. **FLT_DIG:** μέγιστος αριθμός ψηφίων που μπορούν να παρασταθούν πριν γίνει στρογγυλοποίση σε float. Τυπική τιμή 6.

8. **DBL_DIG :** μέγιστος αριθμός ψηφίων που μπορούν να παρασταθούν πριν γίνει στρογγυλοποίση σε double. Τυπική τιμή 15.
9. **LDBL_DIG :** μέγιστος αριθμός ψηφίων που μπορούν να παρασταθούν πριν γίνει στρογγυλοποίση σε longdouble. Τυπική τιμή 18.

10.9. Η ΒΙΒΛΙΟΘΗΚΗ iso646.h

Εδώ ορίζονται 11 μακροεντολές, οι εξής:

1. and : ορίζεται ως &&
2. and_eq : ορίζεται ως &=
3. bitand : ορίζεται ως &
4. bitor :ορίζεται ως |
5. compl : ορίζεται ως ~
6. not : ορίζεται ως !
7. not_eq : ορίζεται ως !=
8. or : ορίζεται ως ||
9. or_eq : ορίζεται ως |=
10. xor : ορίζεται ως ^
11. xor_eq : ορίζεται ως ^=

10.10. Η ΒΙΒΛΙΟΘΗΚΗ stdlib.h

10.10.1. Ησυνάρτηση **exit( )**. Ηδήλωσή της είναι:

void exit (intstatus);

Προκαλεί τερματισμό του προγράμματος, η δέ παράμετρός της καθορίζει τον τρόπο τερματισμού. Εάν η τιμή αυτή είναι 0, θεωρούμε ότι το πρόγραμμα τερματίστηκε κανονικά.

10.10.2. Η συνάρτηση **bsearch( )**. Εκτελεί δυαδική αναζήτηση σε ένα πίνακα, αναζητώντας μια τιμή. Ο πίνακας πρέπει να είναι ταξινομημένος. Η δήλωσή της είναι:

void *bsearch(void *key, void *addr, size_t num, size_t size, int (*cmp) (const void *item1, const void *item2));

Η συνάρτηση επιστρέφει ένα δείκτη στο στοιχείο του πίνακα όπου βρέθηκε η αναζητούμενη τιμή ή NULL, εάν η τιμή δεν υπάρχει στον πίνακα. Στην παραπάνω δήλωση:

- **key** : η τιμή την οποία αναζητούμε.
- **addr** : η διεύθυνση ενός ταξινομημένου πίνακα, στον οποίο διεξάγουμε την αναζήτηση.
- **num** : το πλήθος των στοιχείων, στα οποία διεξάγουμε την αναζήτηση.
- **size** : το μέγεθος των στοιχείων σε byte.
- **cmp**: δείκτης σε μια συνάρτηση, η οποία δέχεται σαν παραμέτρους δύο δείκτες σε δύο στοιχεία του πίνακα, συγκρίνει τις τιμές τους και επιστρέφει ακέραια τιμή. Η τιμή αυτή είναι <0 εάν *item1<*item2, ==0 εάν *item1==*item2 και >0 εάν *item1>*item2.

Στο πρόγραμμα που ακολουθεί γίνεται δυαδική αναζήτηση του αριθμού 32 σε ένα πίνακα ακεραίων.

```
int cmpfunc (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b ); }

int values[ ] = { 5, 20, 29, 32, 63 };

int main ( ) {
    int *item;
    int key = 32;
    int (*cmp) (const void *, const void*);
    cmp = &cmpfunc;
    item = (int*) bsearch (&key, values, 5, sizeof (int), cmp);
    if ( item != NULL )
        printf("Found item = %d\n", *item);
    else
        printf("Item = %d could not be found\n", *item);
    return(0);}
```

10.10.3. Η συνάρτηση **qsort( )**. Εκτελεί ταξινόμηση των στοιχείων ενός πίνακα. Η δήλωσή της είναι:

```
void *qsort (void *base, size_t num, size_t size, int  (*cmp) (const
                    void *item1, const void *item2));
```

Στην παραπάνω δήλωση:

- **base** :ο πίνακας τον οποίο θα ταξινομήσουμε.
- **num** : το πλήθος των στοιχείων, στα οποία διεξάγουμε την ταξινόμηση.
- **size** : το μέγεθος των στοιχείων σε byte.
- **cmp**: ό,τι και στην συνάρτηση bsearch( ).

Στο πρόγραμμα που ακολουθεί γίνεται ταξινόμηση του πίνακα values, στη συνέχεια δε εμφανίζονται τα στοιχεία του στην οθόνη.

```
#define N 10

int cmpfunc (const void * a, const void * b) {
     return ( *(int*)a - *(int*)b ); }

int values[ ] = { 5, -10, 3, 29, -3, 12, 6 };

int main ( ) {
     int k;
     int (*cmp) (const void *, const void*);

     cmp = &cmpfunc;
     qsort (values, 7, sizeof (int), cmp);
     for (k=0; k<7; k++)
          printf("%d", values[k]);
     return(0);}
```

10.10.4. Η συνάρτηση **rand( )**. Εεπιστρέφει ένα τυχαίο θετικό ακέραιο μεταξύ 0 και RAND_MAX (τυπική τιμή +32767). Η δήλωσή της είναι:

```
int  rand ( );
```

Τους ακέραιους τους δημιουργεί με την χρήση μιας γεννήτριας παραγωγής ψευδοτυχαίων αριθμών. Πριν την χρήση της καλούμε τη συνάρτηση srand( ) ως εξής:

```
srand (time (NULL));
```

10.10.5. Οι συναρτήσεις μετατροπής συμβολοσειράς σε double, integer, long integer, long long integer. Είναι κατά σειρά οι:**atof( )**, **atoi( )**, **atol( )**, **atoll( )**.

10.10.6. Ησυνάρτηση **strtod( )**. Η δήλωσή της είναι:

```
double  strtod(const char *str, char **ptr);
```

Μετατρέπει τη συμβολοσειρά str σε float τιμή. Εάν το ptr δεν είναι NULL, τοποθετείται ο δείκτης ptr στον πρώτο χαρακτήρα της str μετά το αριθμητικό μέρος. Το παρακάτω πρόγραμμα:

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

int main ( ) {
     char str[30] = "12.3563This is a value";
     char *ptr;
     double ret;

     ret = strtod(str, &ptr);
     printf ("The number(double) is %lf\n", ret);
     printf ("String part is <%s>", ptr);
     return(0); }
```

θα εμφανίσει στην οθόνη:

The number(double) is 12.356300
String part is <This is a value>

Αντίστοιχες είναι οι συναρτήσεις **strtof( )** και **strtold( )**.

10.10.7. Ησυνάρτηση **strtol( )**. Η δήλωσή της είναι:

long int strtol(const char *str, char **ptr, int base);

str : η παράσταση του ακεραίου αριθμού.

ptr: ο δείκτης ptr στον πρώτο χαρακτήρα της str μετά το αριθμητικό μέρος.

base: το αριθμητικό σύστημα στο οποίο είναι εκφρασμένος ο αριθμός στο str. Τιμή μεταξύ 2 και 36.

Το παρακάτω πρόγραμμα:

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

int main ( ) {
    char str[30] = "1256This is a value";
    char *ptr;
    long ret;

    ret = strtod(str, &ptr, 16);
    printf("The number(long) is %ld\n", ret);
    printf("String part is <%s>", ptr);
    return(0); }
```

θα εμφανίσει στην οθόνη:

The number(long) is 4694
String part is <This is a value>

Αντίστοιχες είναι οι συναρτήσεις **strtoll( )**,**strtoul( )** και**strtoull( )**

10.10.8. Οι συναρτήσεις δυναμικής δέσμευσης μνήμης **malloc( )**, **calloc( )**, **realloc( )** και **free( )**.

10.10.9. Οι συναρτήσεις **abs( )**, **labs( )**, **fabs( )** και **ldiv( )**. Οι δηλώσεις τους είναι:

int abs (int n);

long int labs (long int n);

double fabs (double x);

10.10.10. Ησυνάρτηση **ldiv( )**. Η δήλωσή της είναι:

ldiv_t ldiv (long int numer, long int denum);

Υπολογίζει το ακέραιο πηλίκο και το υπόλοιπο της διαίρεσης numer/denom. Επιστρέφει μια δομή του τύπου ldiv_t. Αυτή η δομή έχει δύο πεδία τύπου long, τα q (κρατάει το πηλίκο) και r (κρατάει το υπόλοιπο).

10.11. Η ΒΙΒΛΙΟΘΗΚΗ stdio.h

10.11.1. Η σταθερά **size_t**. Ακέραιου τύπου. Είναι το αποτέλεσμα του τελεστή **sizeof**.

10.11.2. Ο τύπος **FILE**. Για την αποθήκευση πληροφοριών για ροές.

10.11.3. Ο τύπος **fpos_t**. Για την αποθήκευση θέσης αρχείου.

10.11.4. Η μακροεντολή **NULL**. Καθορίζει την τιμή για ένα null δείκτη.

10.11.5. Η μακροεντολή **EOF**. Αρνητικός ακέραιος για τον καθορισμό του τέλους ενός αρχείου.

10.11.6. Η μακροεντολή **FOPEN_MAX**. Ακέραιοςπου ισούται με το μέγιστο αριθμό των αρχείων, τα οποία μπορούν να είναι ταυτόχρονα ανοιχτά στο σύστημα.

10.11.7. Η μακροεντολή **FILENAME_MAX**. Ακέραιοςπου ισούται με το μέγιστο μήκος ενός πίνακα χαρακτήρων για την αποθήκευση του μέγιστου δυνατού ονόματος αρχείου.

10.11.8. Οι μακροεντολές **SEEK_CUR**,**SEEK_END** και **SEEK_SET**, οι οποίες χρησιμοποιούνται από την fseek( ) για τον εντοπισμό της θέσης μέσα σε ένα αρχείο.

10.11.9. Οι μακροεντολές **stderr**, **stdin** και **stdout.** Δείκτες στα στάνταρ αρχεία λάθους, εισόδου και εξόδου.

10.11.10. Ένας αριθμός γνωστών συναρτήσεων, οι:

- **int fclose(FILE *);**
- **int feof (FILE *);**
- **int fflush (FILE *);**
- **int fopen (const char*, const char*, FILE *);**
- **size_t fread (void *, size_t, size_t, FILE *);**
- **int fseek (FILE *, long int, int);**

- **long int ftell (FILE *);**
- **size_t fwrite (const void *, size_t, size_t, FILE *);**
- **int remove (const char *);**
- **void rewind (FILE *);**
- **int printf (const char*, …);**
- **int fprintf (FILE *, const char *, …);**
- **int scanf (const char *, …);**
- **int fscanf (const char *, const char *, …);**
- **int fgetc (FILE *);**
- **char *fgets (char *, int, FILE *);**
- **int fputc (char, FILE *);**
- **int fputs (const char *, FILE *);**
- **int getc (FILE *);**
- **int getchar (void);**
- **char *gets (char *);**
- **int putc (int, FILE *);**
- **int putchar (int);**
- **int puts (const char *);**
- **void perror (const char *);**
- **int ferror (FILE *);**

10.11.11. Η συνάρτηση **fgetpos( )**.

int fgetpos (FILE *, fpos_t);

Γράφει την τρέχουσα θέση του δείκτη θέσης αρχείου στο δεύτερο όρισμά της.

10.11.12. Η συνάρτηση **rename( )**.

int rename (const char*, const char*);

Αλλάζει το όνομα του πρώτου της ορίσματος στο δεύτερο όρισμα.

10.11.13. Η συνάρτηση **sprintf( )**.

int sprintf (char*, const char*);

Όπως η fprintf( ) αλλά γράφει στο πρώτο της όρισμα και όχι στο stdout.

10.11.14. Η συνάρτηση **tmpfile()**.

FILE *tmpfile (void);

Δημιουργεί ένα προσωρινό δυαδικό αρχείο (wb+).

10.11.15. Η συνάρτηση **clearerr()**.

void clearerr (FILE *);

Καταργεί το eof από το αρχείο.

10.11.16. Η συνάρτηση **tmpnam()**.

char * tmpnam (char *);

Δημιουργεί και επιστρέφει ένα έγκυρο όνομα αρχείου, το οποίο δεν υπάρχει.

ΚΕΦΑΛΑΙΟ 11
ΠΟΛΥΝΗΜΑΤΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

11.1. ΓΕΝΙΚΑ.

Στα πρώτα χρόνια λειτουργίας των υπολογιστών, αυτοί είχαν τη δυνατότητα να τρέχουν ένα πρόγραμμα κάθε φορά. Τέτοια συστήματα ήταν γνωστά ως *single tasking* ή *single processing*. Εμφανίστηκαν στην συνέχεια λειτουργικά συστήματα ικανά να τρέχουν περισσότερα από ένα προγράμματα «ταυτόχρονα», με την διαδικασία της χρήσης χρονικών παραθύρων. Αυτά είναι γνωστά ως *time sharing* λειτουργικά συστήματα. Με τα λειτουργικά συστήματα που εμφανίστηκαν κατόπιν (π.χ. UNIX) μπορούμε να τρέχουμε πολλά προγράμματα ταυτόχρονα σε μια μηχανή, καθένα από τα οποία αποτελεί μια διεργασία (process). Πάντως, ακόμη και στα συστήματα πολλαπλών διεργασιών (*multi processing*), υπάρχει το θέμα ότι κάθε διεργασία μπορεί να εκτελεί μια μόνο ενέργεια κάθε στιγμή. Ένας τρόπος να ξεπεραστεί αυτό το πρόβλημα είναι ο «*πολυνηματικός προγραμματισμός*». Σε αυτόν, μια διεργασία μπορεί να κάνει περισσότερα από ένα πράγματα ταυτόχρονα.

Ένα νήμα είναι μια ακολουθία εντολών μέσα σε μια διεργασίακαι η οποία είναι ρυθμισμένη από το λειτουργικό σύστημα να τρέχει ανεξάρτητα. Θα μπορούσε δηλαδή κανείς να την θεωρήσει ως μια «ελαφρά» διεργασία. Κάθε νήμα έχει ένα μοναδικό αριθμό, ο οποίος το χαρακτηρίζει, μια ταυτότητα δηλαδή, την δικιά του στοίβα (stack), τον δικό του μετρητή προγράμματος (program counter) και τους δικούς του καταχωρητές, μοιράζεται όμως τον ίδιο χώρο μνήμης με άλλα νήματα, ενώ οι διεργασίες (processes) που εκτελούνται σε ένα σύστημα χρησιμοποιούν κάθε μια την δική της μνήμη. Μια ομάδα νημάτων εκτελείται μέσα στην ίδια διεργασία. Τα νήματα αυτά εκτελούνται κατά κάποιον τρόπο παράλληλα, χρησιμοποιώντας χρονικά «παράθυρα» ή πραγματικά παράλληλα εάν το σύστημα έχει πολλούς επεξεργαστές. Κάποιοι λόγοι υπέρ της χρήσης νημάτων είναι:

- Η δημιουργία νημάτων είναι πολύ γρήγορη, καθώς και η εναλλαγή και η επικοινωνία μεταξύ τους.
- Μεγαλύτερη αποτελεσματικότητα, αφού για παράδειγμα ένα νήμα μπορεί να χειρίζεται είσοδο και έξοδο δεδομένων, ενώ ένα άλλο εκτελεί υπολογισμούς στο παρασκήνιο.
- Πιο αποτελεσματική χρήση των πόρων του συστήματος, π.χ. της μνήμης και του χρόνου απασχόλησης της CPU.

- Η δυνατότητα εκτέλεσής τους σε συστήματα ενός, όπως και πολλαπλών επεξεργαστών.
- Γρήγορη επικοινωνία μεταξύ των νημάτων.
- Εύκολος τερματισμός.
- Πιο δομημένα προγράμματα.

Η χρήση πάντως των νημάτων δεν είναι χωρίς προβλήματα. Υπάρχει για παράδειγμα πρόβλημα όταν δυο νήματα επιχειρούν να προσπελάσουν τους ίδιους πόρους του συστήματος και η χρήση κοινής μνήμης μεταξύ τους. Επιπλέον, η διόρθωση (debugging) προγραμμάτων που χρησιμοποιούν νήματα, είναι κάθε άλλο παρά εύκολη υπόθεση. Για τους λόγους αυτούς, πολλά προγράμματα προτιμούν υλοποιήσεις με πολλαπλές διεργασίες.

Το στάνταρ API για τα νήματα είναι το POSIX, ή Pthreads. Για να χρησιμοποιήσουμε τα Pthreads σε Linux για ένα αρχείο που λέγεται test.c, κάνουμε τα παρακάτω:

#include<pthread.h>

και για την μεταγλώττιση και εκτέλεση:

gcc -otesttest.c –lpthread

Κάθε διεργασία, από τη στιγμή που καλείται η main( ), αποτελείται από τουλάχιστον ένα νήμα, το οποίο μπορεί να δημιουργήσει επιπλέον νήματα.

Θα ξεκινήσουμε λοιπόν με τη μελέτη της βιβλιοθήκης **pthread.h**

11.2. ΔΗΜΙΟΥΡΓΙΑ ΝΗΜΑΤΟΣ.

Η δημιουργία νήματος γίνεται με τη χρήση της συνάρτησης **pthread_create**. Η επικεφαλίδα της συνάρτησης είναι η εξής:

int pthread_create (pthread_t *id, pthread_attr_t *attr, void *(*thread_f)(void *), void *arg);

Σχόλια:

- Για κάθε νήμα απαιτείται η «ταυτότητά» του, η οποία δίνεται ως πρώτο όρισμα στη συνάρτηση. Για την ακρίβεια, δίνεται ένας δείκτης προς το νήμα. Το **pthread_t** είναι τύπος μη προσημασμένου ακεραίου και είναι γνωστό μόνο μέσα στη συγκεκριμένη διεργασία.
- Το δεύτερο όρισμά καθορίζει ιδιότητες (attributes) της συνάρτησης. Αν το όρισμα αυτό είναι NULL, οι ιδιότητες είναι κάποιες προκαθορισμένες (default) συνήθως δε, δεν χρειάζεται κάποια άλλη τιμή. Οι διαφορετικές περιπτώσεις

έχουν να κάνουν με την δημιουργία «αυτόνομων» νημάτων, τα οποία δεν συνδέονται με τα υπόλοιπα, με την αλλαγή του μεγέθους της stack του νήματος κλπ.

- Το τρίτο όρισμα είναι δείκτης σε μια συνάρτηση, η οποία παίρνει όρισμα δείκτη σε void (σε ο,τιδήποτε δηλαδή) και επιστρέφει δείκτη σε void (ο,τιδήποτε). Η συνάρτηση αυτή είναι εκείνη που θα εκτελεστεί όταν δημιουργηθεί το νήμα.
- Με το τέταρτο όρισμα μεταβιβάζουμε τιμές στην συνάρτηση του νήματος. Αυτή η παράμετρος είναι μόνο μία. Αν θέλουμε να περάσουμε περισσότερες παραμέτρους, τις τοποθετούμε σε μια structure και δίνουμε στη θέση αυτή την διεύθυνσή της.
- Η συνάρτηση **pthread_create( )** επιστρέφει 0 εάν το νήμα δημιουργηθεί σωστά και μη μηδενική τιμή σε περίπτωση αποτυχίας.

Παράδειγμα 1.

Παρουσιάζουμε ένα παράδειγμα δημιουργίας ενός απλού νήματος:

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

int ak;

void *one(void *);

int main( ) {
    pthread_t tarr;
    if (pthread_create(&tarr, NULL, &one, NULL)) {
        printf ("Unable to create thread");
        exit(1); }
    printf ("%d\n", ak);
    sleep(10);
    printf ("Done\n");
    return 0;}

void *one (void *) {
    printf ("I am the child thread\n");
    sleep(3);
    ak = 15;
    printf ("%d\n", ak);
    pthread_exit( 0 ); }
```

Σχόλια στο παραπάνω πρόγραμμα:

- Η βιβλιοθήκη **pthread.h** περιέχει, όπως έχουμε ήδη αναφέρει τις συναρτήσεις που απαιτούνται για την δημιουργία πολυνηματικών εφαρμογών.

- Το **unistd.h** χρειάζεται για την συνάρτηση **sleep( )**. Αυτή εισάγει καθυστέρηση στο πρόγραμμα τόσων δευτερολέπτων, όσο είναι το όρισμά της.
- Με την **pthread_create( )**δημιουργείται το νήμα. Η main( ) γράφει αρχικά στην οθόνη την τιμή 0, αφού η ak είναι εξωτερική μεταβλητή και με τη sleep(10) εισάγεται μια καθυστέρηση 10 δευτερολέπτων στο κυρίως πρόγραμμα. Ενώ ακόμη διαρκούν τα 10 δευτερόλεπτα, το νήμα one( ) γράφει στην οθόνη τη φράση "I am the child thread" περιμένει για 3 δευτερόλεπτα, αλλάζει την τιμή του ak σε 15 και γράφει αυτή την τιμή στην οθόνη. Το νήμα τερματίζεται με την **pthread_exit( )**. Όταν τελειώσουν τα 10 δευτερόλεπτα, από τη main( ) γράφεται "Done" και το πρόγραμμα τελειώνει.

Παράδειγμα 2.

Στο πρόγραμμα που ακολουθεί χρησιμοποιούμεένα πίνακα νημάτων, τον tarr:

```
#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#define THREADS 5

void *two (void *);

int main() {
   int i;
   pthread_t tarr[THREADS];

   for (i=0; i<THREADS; i++) {
        if (pthread_create (&tarr[i], NULL, &two, NULL)) {
            printf ("Error in creating thread\n");
            exit(1); }
       printf ("ProcessID:%d Created thread's ID:%u\n",
                getpid( ), (int)tarr[i]);}
   sleep(5);
   return 0; }

void *two(void *) {
   printf ("This is the thread no %u\n", pthread_self());
   pthread_exit( 0 ); }
```

Η έξοδος του προγράμματος θα είναι κάτι όπως το παρακάτω:

```
Process ID: 2112  Created thread's ID: 1
This is the thread no 1
This is the thread no 2
Process ID: 2112  Created thread's ID: 2
Process ID: 2112  Created thread's ID: 3
This is the thread no 3
Process ID: 2112  Created thread's ID: 4
This is the thread no 4
Process ID: 2112  Created thread's ID: 5
This is the thread no 5
```

Σχόλια στο παραπάνω πρόγραμμα:

- Η συνάρτηση **getpid( )** επιστρέφει τον αριθμό (την «ταυτότητα») της τρέχουσας διεργασίας.
- Η δήλωση της συνάρτησης **pthread_self( )** είναι η εξής:

  pthread_t pthread_self(void);

  Η συνάρτηση επιστρέφει την ταυτότητα του νήματος μέσα από το οποίο έχει κληθεί.

  Παρατηρείστε πότε εμφανίζονται τα μηνύματα στην οθόνη. Κάποια από τα νήματα έχουν τελειώσει την λειτουργία τους πριν εμφανιστεί το μήνυμα δημιουργίας των, όχι επειδή (παραδόξως) δεν έχουν ήδη δημιουργηθεί, αλλά αφού η εκτέλεση της printf( ) είναι χρονοβόρα.

Παράδειγμα 3.

Το πρόγραμμα είναι μια παραλλαγή του προηγουμένου. Χρησιμοποιούμε και πάλι ένα πίνακα νημάτων. Τα νήματα αυτά δημιουργούνται με τη σειρά 1^{ο}, 2^{ο}, 3^{ο}, αλλά σε καθένα από αυτά εισάγεται διαφορετική καθυστέρηση. Ανάλογα με αυτή εμφανίζονται και τα μηνύματα τερματισμού κάθε νήματος:

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

#define THREADS 3

void *one(void *);
void *two(void *);
void *three(void *);

int main() {
      int i;
      pthread_t tarr[THREADS];

      if (pthread_create( &tarr[0], NULL, &one, NULL)) {
            printf ("Error in creating thread one\n");
            exit(1); }
```

```
        if (pthread_create( &tarr[1], NULL, &two, NULL)) {
              printf ("Error in creating thread two\n");
              exit(1); }
        if (pthread_create( &tarr[2], NULL, &three, NULL)) {
              printf ("Error in creating thread three\n");
              exit(1); }
        printf ("ProcessID:%d Created thread's ID:%u\n",
                 getpid( ), (int) tarr[i]);
        sleep(35);
        return 0;}

    void *one (void *) {
        printf ("Thread %u\n", pthread_self());
        sleep(20);
        printf ("Thread one(1) finishes\n");
        pthread_exit( 0 ); }

    void *two (void *) {
        printf ("Thread %u\n", pthread_self());
        sleep(10);
        printf ("Thread two (2) finishes\n");
        pthread_exit( 0 ); }

    void *three (void *) {
        printf ("Thread %u\n", pthread_self());
        sleep(15);
        printf ("Thread three (3) finishes\n");
        pthread_exit( 0 ); }
```

Σχόλια στο παραπάνω πρόγραμμα:

Η έξοδος του προγράμματος θα είναι η παρακάτω:

```
Thread 1
Thread 2
Thread 3
Thread two (2) finishes
Thread three (3) finishes
Threadone (1) finishes
```

Παράδειγμα 4.

Το παράδειγμα αυτό είναι μια ακόμα παραλλαγή των προηγουμένων. Εδώ στην συνάρτηση περνάμε όρισμα, το οποίο δίδεται μέσω της 4^{ης} παραμέτρου της pthread_create( ). Χρησιμοποιούμε την ίδια συνάρτηση και για τα τρία νήματα, δίνοντας όμως ως παράμετρο κάθε φορά τον μετρητή του for:

```
#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#define THREADS 3

void *test (void *);
```

```
int main( ) {
    int i;
    pthread_t tarr[THREADS];
    for (i=0; i<THREADS; i++) {
        pthread_create (&tarr[i], NULL, &test, (void *)&i);
        printf ("ProcessID:%d Created thread's ID:%u\n",
            getpid( ), (int)tarr[i]); }
    sleep(5);
    return 0; }

void *test(void *ptr) {
    printf ("This is the thread no %u\n", pthread_self( ));
    printf ("or else %d\n", *((int *)ptr));
    pthread_exit (0); }
```

Η έξοδος του προγράμματος θα είναι κάτι όπως το παρακάτω:

```
Process ID: 4108  Created thread's ID: 1
This is the thread no 1
or else 1
This is the thread no 2
or else 1
Process ID: 4108  Created thread's ID: 2
Process ID: 4108  Created thread's ID: 3
This is the thread no 3
or else 3
```

Παρατηρούμε τα εξής:

- Το 4ο όρισμα της pthread_create( ) είναι δείκτης σε void. Για τον λόγο αυτόν γίνεται προσαρμογή τύπου του δείκτη &i ο οποίος είναι δείκτης σε ακέραιο, οπότε συμβαδίζει και με τον τύπο του δείκτη ptr, ο οποίος είναι όρισμα της test( ).
- Προκειμένου τα περιεχόμενα του δείκτη ptr να εμφανιστούν ως ακέραια τιμή μέσα στην test( ), γίνεται και πάλι προσαρμογή τύπου του ptr σε δείκτη σε ακέραιο, τα δε πειεχόμενά του (ακέραιος) εμφανίζονται με τον προσδιοριστή %d.

Παράδειγμα 5.

Στη συνάρτηση three στο παρακάτω «περνάμε» δύο ορίσματα. Αυτό γίνεται κατορθωτό μέσω μιας structure τύπου COMP, η οποία «ενσωματώνει» τις δύο τιμές, τις οποίες θέλουμε να περάσουμε στην συνάρτηση. Σε κάθε νήμα περνάει μια structure, η οποία έχει ως ακέραιο πεδίο ένα τυχόντα θετικό ακέραιο μεταξύ 0 και 19 και μια συμβολοσειρά την οποία δίνει ο χρήστης. Κάθε νήμα εισάγει καθυστέρηση τόσων δευτερολέπτων, όσο το ακέραιο

όρισμα που μεταβιβάστηκε στην συνάρτησή του. Στο τέλος, η main( ) τερματίζεται μετά από 30 δευτερόλεπτα, στη διάρκεια των οποίων έχουν τερματίσει όλα τα νήματα:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>


#define THREADS 3


typedef struct comp{
      int ak;
      char pin[15]; } COMP;


void *three (void *);


int main( ) {
      int i;
      COMP  examp [THREADS];
      pthread_t  tarr [THREADS];
      char  temp [15];

      srand(time(NULL));

      for (i=0; i<THREADS; i++) {
            examp[i].ak = rand( )%20;
            scanf("%s", examp[i].pin);
            fflush(stdin); }

      for (i=0; i<THREADS; i++)
            printf("%d %s\n", examp[i].ak, examp[i].pin);

      for (i=0; i<THREADS; i++) {
            pthread_create(&tarr[i], NULL, &three, (void *)&examp[i]);
            printf("ProcessID:%d Created thread's ID:%u\n",
                  getpid( ), (int)tarr[i]);}
      return 0; }


void *three (void *ptr) {
      printf ("This is the thread no %u\n", pthread_self( ));
      sleep (((COMP *)ptr)->ak);
      printf ("%d and %s\n", ((COMP *)ptr)->ak, ((COMP *)ptr)->pin);
      pthread_exit (0); }
```

Με δεδομένο ότι ο χρήστης δίνει τις συμβολοσειρές One Two και Three απο το πληκτρολόγιο, η έξοδος του προγράμματος θα είναι κάτι όπως το παρακάτω:

3 One
11 Two
5 Three
Process ID: 3916 Created thread's ID: 1
This is the thread no 1
Process ID: 3916 Created thread's ID: 2
Process ID: 3916 Created thread's ID: 3
This is the thread no 3

Παρατηρείστε ότι τα νήματα δεν τέλειωσαν την εργασία τους, παρ' όλα αυτά το πρόγραμμα τερματίστηκε, αφού τελείωσε η main( ). Θα αναφερθούμε ξανά σε αυτό πιο κάτω, μιλώντας για νήματα σε αναμονή.

11.3. ΤΕΡΜΑΤΙΣΜΟΣ ΝΗΜΑΤΟΣ.

Ο τερματισμός ενός νήματος μπορεί να γίνει με διάφορους τρόπους. Αρχικά, εάν από ένα νήμα κληθεί η συνάρτηση exit( ), τότε διακόπτεται ολόκληρη η διεργασία, άρα και όλα τα νήματα.

Ένα μόνο νήμα μπορεί να διακοπεί:

- Με return( ) μέσα στο νήμα.
- Με την κλήση της συνάρτησης:

  void pthread_exit (void *ret);

  στην οποία το ret αντιστοιχεί σε κωδικό λάθους (0 εάν δε υπάρχει λάθος). Η κλήση της στο τέλος της main( ) έχει ως συνέπεια να περιμένει η main( ) να τερματιστούν όλα τα νήματα.
- Με την κλήση της συνάρτησης:

  int pthread_cancel(pthread_t id));

  με την οποία ένα άλλο νήμα της ίδιας διεργασίας διακόπτει το νήμα με ταυτότητα id.

11.4. ΟΡΦΑΝΑ ΝΗΜΑΤΑ.

Υπάρχει περίπτωση ο γονέας, η διεργασία δηλαδή που δημιούργησε κάποια νήματα, να τερματιστεί πριν τα νήματα (ή κάποιο από αυτά) προλάβουν να τελειώσουν και να τερματιστούν. Σε μια τέτοια περίπτωση τα νήματα-παιδιά τερματίζονται και αυτά. Άρα δεν υπάρχουν «ορφανά» νήματα. Κάτι τέτοιο θα συνέβαινε με την εκτέλεση του

παρακάτω προγράμματος (έχουν παραλειφθεί για συντομία τα include στην αρχή του προγράμματος):

```
#define THREADS 3
void *three (void *);
int main( ) {
   int i;
   pthread_t tarr[THREADS];

   for (i=0; i<THREADS; i++) {
        if (pthread_create(&tarr[i], NULL, &three, NULL)) {
             printf("Error in creating thread\n");
             exit(1); }
        printf("Process ID:%d Created thread's ID:%u\n",
                  getpid( ), (int)tarr[i]);}
   return 0; }

void *three (void *) {
   sleep(5);
   printf("This is the thread no %u\n", pthread_self( ));
   pthread_exit( 0 ); }
```

Η sleep(5) τέθηκε στην three( )για να καθυστερήσει, ώστε να ολοκληρώσει η main( ) την εργασία της πριν το νήμα, το οποίο θα τερματιστεί και δεν θα βγάλει κανένα μήνυμα.

11.5. ΝΗΜΑΤΑ ΣΕ ΑΝΑΜΟΝΗ.

Για την αντιμετώπιση ενός προβλήματος όπως το παραπάνω, προκειμένου δηλαδή το πρόγραμμα να περιμένει μέχρι να ολοκληρώσουν τα νήματα την εργασία τους πριν τερματιστεί, χρησιμοποιείται η συνάρτηση **ptherad_join( )**. Η δήλωσή της είναι:

int ptherad_join (ptherad_t, void **);

Το πρώτο της όρισμα είναι το αναγνωριστικό, η ταυτότητα του νήματος. Το δεύτερο όρισμα χρησιμοποιείται για την αποθήκευση της τιμής επιστροφής της συνάρτησης, η οποία καλείται από το νήμα. Η τιμή επιστροφής της συνάρτησης αυτής έχει τιμή επιστροφής, όπως ξέρουμε, void* και αποθηκεύεται στον χώρο που δείχνεται από το δεύτερο όρισμα, το οποίο λοιπόν είναι τύπου void**.

Το νήμα, το οποίο καλεί την pthread_join( ), αναστέλλει την εκτέλεσή του μέχρι να τερματίσει την λειτουργία του το καλούμενο νήμα. Στο παρακάτω παράδειγμα το νήμα δημιουργείται κανονικά. Η main( ) εμφανίζει το μήνυμα:

Waiting for the thread to end...

ενώ το νήμα λειτουργεί. Στη συνέχεια εκτελείται η pthread_join( ). Η main( ) δεν προχωρεί στην επόμενη εντολή, αλλά περιμένει τα 20 δευτερόλεπτα που απαιτούνται για να τελειώσει το νήμα. Το νήμα τελειώνοντας βγάζει το μήνυμα **Done**, στη συνέχεια η main( ) γράφει **Thread ended.** και το πρόγραμμα τελειώνει.

```
void *wait (void *) {
    sleep(20);
    printf("Done.\n"); }

int main (void) {
    pthread_t thread;
    int err;

    err = pthread_create(&thread, NULL, &wait, NULL);
    if (err) {
        printf("An error occured: %d", err);
        return 1; }
    printf("Waiting for the thread to end...\n");
    pthread_join(thread, NULL);
    printf("Thread ended.\n");
    return 0; }
```

Παρατηρείστε μια παραλλαγή του παραδείγματος 5 της παραγράφου 11.2. Εδώ, η main( ) περιμένει να τερματιστούν όλα τα νήματα πριν ολοκληρωθεί και τερματιστεί το πρόγραμμα. Αυτή η αναμονή γίνεται με την κλήση της pthread_join( ) μέσα στο τελευταίο for.

```
#define THREADS 3
#define NUM 3

typedef struct comp{
    int ak;
    char pin[15]; } COMP;

void *four(void *);

int main( ) {
    int i;
    COMP examp[NUM];
    pthread_t tarr[THREADS];
    char temp[15];

    srand(time(NULL));
    for (i=0; i<THREADS; i++) {
        examp[i].ak = rand( )%20;
        scanf("%s", examp[i].pin);
        fflush(stdin); }

    for (i=0; i<THREADS; i++) {
        pthread_create (&tarr[i], NULL, &four, (void *)&examp[i]);
        printf ("ProcessID:%d Created thread's ID:%u\n",
            getpid( ), (int)tarr[i]); }
```

```
    for (i=0; i<THREADS; i++)
        pthread_join (tarr[i], NULL);
    return 0; }

void *four (void *ptr) {
    printf("This is the thread no %u\n", pthread_self());
    sleep(((COMP *)ptr)->ak);
    printf(" %d and %s\n", ((COMP *)ptr)->ak, ((COMP *)ptr)->pin);
    pthread_exit( 0 ); }
```

Στην οθόνη εμφανίζεται κάτι όπως το παρακάτω. Πριν από τις τρεις τελευταίες γραμμές στην οθόνη, εισάγεται καθυστέρηση σύμφωνα με αυτή που βάζει κάθε νήμα (υποθέτουμε και πάλι ότι ως συμβολοσειρές δίνουμε τις λέξεις One, Two και Three):

```
Process ID: 5796  Created thread's ID: 1
This is the thread no 1
Process ID: 5796  Created thread's ID: 2
This is the thread no 2
Process ID: 5796  Created thread's ID: 3
This is the thread no 3
8 and Three
10 and Two
17 and One
```

11.6. MUTEX.

Το να αναφέρονται πολλές λειτουργίες ταυτόχρονα σε κάποιο πόρο του συστήματος εμπεριέχει κινδύνους. Αν για παράδειγμα τρία νήματα θέλουν να γράψουν ταυτόχρονα σε ένα αρχείο, υπάρχει πρόβλημα, αφού ο δίσκος δουλεύει πολύ πιο αργά από ό,τι μια CPU. Σε τέτοια περίπτωση κλειδώνουμε τα επιπλέον νήματα από αυτό το οποίο γράφει, ώστε να μη δημιουργηθούν συνθήκες επικάλυψης των εγγραφών και άρα λάθη. Ένα mutex (mutual exclusion) είναι μια μεταβλητή και χρησιμοποιείται για αυτόν ακριβώς τον λόγο, αποτρέπει δηλαδή συνθήκες ανταγωνισμού. Πιο συγκεκριμένα, το σύνολο των εντολών ενός νήματος που ελέγχουν, ενημερώνουν κλπ ένα διαμοιραζόμενο πόρο του συστήματος, λέγεται «κρίσιμη περιοχή». Κάθε νήμα πριν εισέλθει στην κρίσιμη περιοχή, προσπαθεί να καταλάβει ένα mutex. Αν ένα άλλο νήμα έχει ήδη εισέλθει στην κρίσιμη περιοχή (άρα χρησιμοποιεί τον συγκεκριμένο πόρο), το mutex είναι κατηλειμμένο και το νήμα που προσπάθησε να μπει στην κρίσιμη περιοχή πρέπει να περιμένει. Μόλις το νήμα καταφέρει να καταλάβει το mutex, εισέρχεται στη κρίσιμη περιοχή, εκτελεί την

ενημέρωση και βγαίνοντας ελευθερώνει το mutex. Το ίδιο εφαρμόζεται σε όλα τα νήματα, τελικά δε η μεταβλητή mutex καταστρέφεται.

Δημιουργία και καταστροφή mutex.

Οι μεταβλητές τύπου mutex δηλώνονται ως τύπου **pthread_mutex_t** και αρχικοποιούνται πριν τη χρήση τους είτε κατά τη δήλωσή τους (με τη χρήση της σταθεράς **PTHREAD_MUTEX_INITIALIZER,** είτε δυναμικά με τη χρήση της συνάρτησης **pthread_mutex_init( ).** Η δεύτερη μέθοδος δίνει τη δυνατότητα να «περάσουν» επιπλέον ιδιότητες στη mutex ως παράμετροι, αν και η τιμή NULL χρησιμοποιείται για τις προκαθορισμένες ιδιότητες.

Η συνάρτηση **pthread_mutex_destroy( )** καταστρέφει ένα mutex όταν τελειώσει η χρήση του.

Κλείδωμα και ξεκλείδωμα mutex.

Μια μεταβλητή mutex κλειδώνεται μέσω της συνάρτησης **pthread_mutex_lock( )**. Αν κάποιο άλλο νήμα έχει ήδη κλειδώσει αυτή τη μεταβλητή, τότε η εκτέλεση του νήματος αναστέλλεται μέχρι να ελευθερωθεί το mutex.

Η συνάρτηση **pthread_mutex_unlock( )** ξεκλειδώνει μια μεταβλητή mutex όταν κληθεί από το νήμα το οποίο την έχει κλειδώσει. Έτσι, ένα άλλο νήμα μπορεί να χρησιμοποιήσει τη mutex. Αν πάντως η mutex είναι ήδη ελεύθερη ή ανήκει σε άλλο νήμα, θα προκληθεί σφάλμα κατά την εκτέλεση.

Δίνουμε στη συνέχεια ένα απλό παράδειγμα χρήσης των mutex.

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

pthread_t tarr[2];
int count;
pthread_mutex_t key;

void *func (void *par) {
        pthread_mutex_lock(&key);
        count++;
        printf("\nNumber %d started \n", count);
        sleep(5);
        printf("\nNumber %d finished \n", count);
        pthread_mutex_unlock(&key);
        return NULL;}

int main(void) {
```

```
int i=0;

pthread_mutex_init(&key, NULL);
while (i<2) {
        pthread_create(&tarr[i], NULL, &func, NULL);
        printf("Thread %d was created\n", tarr[i]);
        i++; }
pthread_join(tarr[0], NULL);
pthread_join(tarr[1], NULL);
pthread_mutex_destroy(&key);
return 0; }
```

Το mutex **key** ορίζεται ως εξωτερική μεταβλητή και αρχικοποιείται με την κλήση της συνάρτησης **pthread_mutex_init( )** ή δίνοντας ως αρχική τιμή του mutex τητν σταθερά **PTHREAD_MUTEX_INITIALIZER** με την εξής εντολή στην main( ):

pthread_mutex_t key = PTHREAD_MUTEX_INITIALIZER;

Στη main( ) δημιουργούνται δύο νήματα, τα tarr[0] και tarr[1]. Στην οθόνη κατά την εκτέλεση θα εμφανιστεί το παρακάτω:

Thread 1 was created
Thread 2 was created
Number 1 started
<Αναμονή 5 δευτερολέπτων>
Number 1 finished
Number 2 started
<Αναμονή 5 δευτερολέπτων>
Number 2 finished

11.7. ΣΗΜΑΦΟΡΟΙ.

Είναι μεταβλητές που χρησιμοποιούνται για σηματοδότηση. Συγκεκριμένα, σε μια ουρά τοποθετούνται διεργασίες που περιμένουν μια συγκεκριμένη τιμή της μεταβλητής, ώστε να ξεκινήσουν να δουλεύουν. Αντίστοιχη χρήση των σημαφόρων γίνεται και στην λειτουργία με νήματα.

Ας δούμε ένα πρόγραμμα παρόμοιο σε ορισμένα σημεία με αυτό του παραδείγματος 2 της παραπάνω παραγράφου 11.2.

```
#define THREADS 5

sem_t  semph;

void *semo (void *ptr) {
   printf("This is the thread no %d\n", *((int *)ptr));
   free (ptr); }

int main( ) {
   int i, *dkt;
   pthread_t tarr[THREADS];
```

```
sem_init (&semph, 0, 1);

for (i=0; i<THREADS; i++) {
      dkt = (int *) malloc (sizeof (int));
      *dkt = i;
      if (pthread_create(&tarr[i], NULL, &semo, dkt)) {
            printf ("Error in creating thread\n");
            exit(1); }

for (i=0; i<THREADS; i++) {
      if (pthread_join (tarr[i], NULL)) {
            printf ("Error in creating thread\n");
            exit(1); }
sem_destroy (&semph);
return 0; }
```

Παρατηρήσεις και σχόλια στο πιο πάνω πρόγραμμα:

- Για την λειτουργία των σημαφόρων εισάγουμε το αρχείο **semaphore.h**.
- Αρχικά ας δούμε πώς δημιουργούμε ένα σημαφόρο και στη συνέχεια θα δούμε πώς ακριβώς «δουλεύει». Δηλώνουμε μια μεταβλητή την semph, η οποία είναι του τύπου **sem_t** (ακέραιος τύπος). Η μεταβλητή αυτή αρχικοποιείται με την χρήση της συνάρτησης **sem_init( )**, ενώ στο τέλος του προγράμματος καλούμε την συνάρτηση **sem_destroy( )** η οποία καταστρέφει τους σημαφόρους που έχουν δημιουργηθεί.

  Η sem_init( ) δέχεται τρία ορίσματα: το πρώτο είναι ένας δείκτης στον σημαφόρο που έχουμε δηλώσει. Το δεύτερο είναι ακέραιος, ο οποίος είναι ίσος με μηδέν όταν αναφερόμαστε σε πολλαπλά νήματα και όχι σε πολλαπλές διεργασίας. Το τρίτο είναι επίσης ένας θετικός ακέραιος (ή πιο σωστά unsigned), ο οποίος δίνει την αρχική τιμή του στον σημαφόρο και η χρήση του οποίου θα φανεί στη συνέχεια. Ας θέσουμε αρχικά την τιμή αυτή ίση με 1.
- Σε ένα σημαφόρο εκτελούνται μόνο δύο λειτουργίες: Η λειτουργία **wait** και η λειτουργία **post**. Αυτές οι δύο λειτουργίες είναι παρόμοιες με τις λειτουργίες lock και unlock των mutex.
- Εδώ (στο παραπάνω πρόγραμμα δηλαδή) πρακτικά δεν χρησιμοποιούνται σημαφόροι και η έξοδος είναι κάτι όπως το παρακάτω, με ένα μήνυμα από κάθε νήμα:

```
This is the thread no 0
This is the thread no 1
This is the thread no 2
This is the thread no 3
This is the thread no 4
```

- Ας τροποποιήσουμε την semo( ) ως εξής:

```
void *semo (void *ptr) {
    sleep(3);
    printf("This is the thread no %d\n", *((int *)ptr));
    free (ptr); }
```

Η έξοδος είναι κάτι όπως το παρακάτω, μετά την αναμονή των 3 δευτερολέπτων που εισάγεται στη semo ( ):

```
<Αναμονή 3 δευτερολέπτων>
This is the thread no 1
This is the thread no 3
This is the thread no 0
This is the thread no 2
This is the thread no 4
```

- Τροποποιούμε ξανά την semo( ) ως εξής:

```
void *semo (void *ptr) {
    sem_wait (&semph);
    sleep(3);
    printf("This is the thread no %d\n", *((int *)ptr));
    sem_post (&semph);
    free (ptr); }
```

Η έξοδος είναι όπως το παρακάτω:

```
<Αναμονή 3 δευτερολέπτων>
This is the thread no 0
<Αναμονή 3 δευτερολέπτων>
This is the thread no 1
<Αναμονή 3 δευτερολέπτων>
This is the thread no 2
<Αναμονή 3 δευτερολέπτων>
This is the thread no 3
<Αναμονή 3 δευτερολέπτων>
This is the thread no 4
```

άρα μόνο ένα νήμα μπορεί να εκτελέσει την συνάρτηση κάθε στιγμή.

- Εξηγούμε την παραπάνω συμπεριφορά, μιλώντας για τον ρόλο της τιμής του σημαφόρου και τη λειτουργία της **sem_wait( )** και της **sem_post( )**:
  - Η **sem_wait( )** ελέγχει την τιμή του σημαφόρου. Εάν αυτή είναι 0, η εκτέλεση του νήματος περιμένει σε εκείνο το σημείο. Εάν είναι η τιμή του σημαφόρου είναι θετική, τότε δεν υπάρχει αναμονή, η τιμή του μειώνεται κατά 1 και το νήμα εκτελεί τις εντολές.
  - Η **sem_post( )** κάνει το αντίστροφο, δηλαδή αυξάνει κατά 1 την τιμή του σημαφόρου.

Με βάση τα παραπάνω, ας παρατηρήσουμε την εκτέλεση του προγράμματος. Συμβολίζουμε με s τον σημαφόρο. Θυμίζουμε ότι του έχει δοθεί αρχική τιμή 1. Η κρίσιμη περιοχή περιλαμβάνει τις εντολές sleep(3) και την printf( ), η οποία γράφει το μήνυμα στην οθόνη. Οι εντολές μεταξύ των sem_wait( ) και sem_post( ) εκτελούνται από ένα μόνο νήμα κάθε φορά.

Νήμα	**s μετά την sem_wait( )**	**Κρίσιμη περιοχή**	**s μετά την sem_post( )**
N0	**0**	**s==0 printf(...)**	**1**
N1	**0**	**s==0 printf(...)**	**1**
N2	**0**	**s==0 printf(...)**	**1**
N3	**0**	**s==0 printf(...)**	**1**
N4	**0**	**s==0 printf(...)**	**1**

- Τροποποιούμε την κλήση της sem_init( ), δίνοντας αρχική τιμή 2 στον σημαφόρο, δηλαδή: **sem_init (&semph, 0, 2);** Τώρα η έξοδος του προγράμματος θα είναι κάτι όπως το παρακάτω:

<Αναμονή 3 δευτερολέπτων>
This is the thread no 1
This is the thread no 3
<Αναμονή 3 δευτερολέπτων>
This is the thread no 0
This is the thread no 2
<Αναμονή 3 δευτερολέπτων>
This is the thread no 4

Δηλαδή δύο από τα νήματα αρχίζουν να τρέχουν, μετά άλλα δύο και τέλος το ένα νήμα που μένει. Για χάρη ευκολίας, ας θεωρήσουμε ότι εκτελούνται κατά σειρά τα νήματα N0, N1, N2, N3, N4. Με την εκτέλεση του N0 το s από 2 γίνεται 1, πράγμα που επιστρέπει και στο N1 να εκτελεστεί στον ίδιο χρόνο, κάνοντας το s ίσο με 0. Ακολουθούν δύο αυξήσεις του s, λόγω του τερματισμού των N0 και N1, οπότε το s ξαναγίνεται 2. Μειώνεται ξανά αρχικά με την εκτέλεση των N2 και N3 και γίνεται 0, ενώ με τον τερματισμό των το s γίνεται ξανά 2. Τέλος μειώνεται ξανά από το N4 για να αυξηθεί στο τέλος της εκτέλεσής του. Αντίστοιχα, αν η αρχική τιμή του σημαφόρου ήταν 3, θα βλέπαμε τρία νήματα να εκτελούνται μαζί αρχικά και μετά τα υπόλοιπα δύο.

Τελειώνοντας, ας υπενθυμίσουμε ότι στα mutex μπορούμε να έχουμε κλείδωμα ενός mutex περισσότερες φορές σε ένα νήμα, αλλά δε μπορούμε να έχουμε περισσότερες φορές κλείδωμα ενός mutex σε διαφορετικά νήματα.