

ΜΥΥ502

Προγραμματισμός Συστημάτων



Β. Δημακόπουλος

dimako@cse.uoi.gr

<http://www.cse.uoi.gr/~dimako>

❖ Αντικείμενο μαθήματος:

- Εκμάθηση βασικών εργαλείων, τεχνικών και μεθόδων για προχωρημένο προγραμματισμό που στοχεύει περισσότερο στο «σύστημα» παρά στην «εφαρμογή»
- Η γλώσσα C είναι μονόδρομος στον προγραμματισμό συστημάτων
 - ✧ Με μεγάλη προγραμματιστική βάση και στο χώρο των εφαρμογών
 - ✧ Προγραμματισμός συστήματος:
 - «κάτω» από το επίπεδο εφαρμογών (υποστηρίζει τις εφαρμογές)
 - Ανάμεσα στις εφαρμογές και το hardware
 - π.χ. μεταφραστές, λειτουργικά συστήματα, συστήματα υποστήριξης εκτέλεσης, ενσωματωμένα συστήματα κλπ.
- Το UNIX/POSIX και τα «POSIX-οειδή» περιβάλλοντα είναι η σημαντικότερη και πιο ολοκληρωμένη πλατφόρμα για εργασία σε επίπεδο συστήματος
 - ✧ Βασικές γνώσεις χρήσιμες και σε επόμενα μαθήματα (π.χ. λειτουργικά συστήματα, παράλληλα συστήματα, μεταφραστές κλπ)

❖ 'Υλη μαθήματος (από τον οδηγό σπουδών):

- “ Η γλώσσα προγραμματισμού C: στοιχειώδης C (βασικοί τύποι δεδομένων, εκφράσεις, τελεστές, δομές ελέγχου ροής, συναρτήσεις), προχωρημένα στοιχεία (πίνακες, δείκτες, δομές), δυναμική διαχείριση μνήμης, είσοδος/έξοδος, προεπεξεργαστής. ”
- “ Βασικές κλήσεις UNIX (διεργασίες, I/O, σήματα). Διαδιεργασιακή επικοινωνία (κοινόχρηστη μνήμη, sockets). Εισαγωγή στον παράλληλο προγραμματισμό (νήματα, mapReduce). Προχωρημένα θέματα (ασφάλεια, γλώσσα μηχανής, εργαλεία ανάπτυξης μεγάλων προγραμμάτων). ”

❖ Δύο μέρη:

➤ Γλώσσα προγραμματισμού C

- ✧ Υποθέτει γνώση προγραμματιστικών τεχνικών
- ✧ Υποθέτει γνώση γλωσσών προγραμματισμού «συγγενών» με την C (π.χ. Java)
- ✧ Καλύπτονται από τις «Τεχνικές Αντικειμενοστραφούς Προγραμματισμού» και «Ανάπτυξη Λογισμικού»

Π1: Περίπου 50% της ύλης

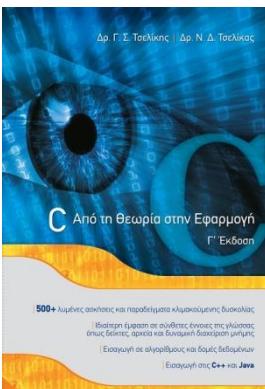
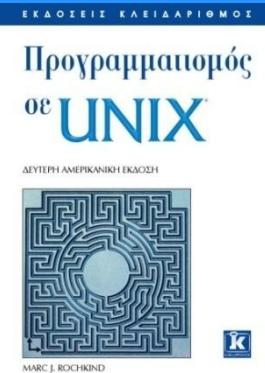
➤ Προγραμματισμός συστημάτων POSIX και προχωρημένα θέματα

- ✧ Εμβάθυνση σε προχωρημένες δυνατότητες της C
- ✧ Γνωριμία με διαδικασίες και εργαλεία ανάπτυξης εφαρμογών συστήματος
- ✧ Βασικές κλήσεις POSIX (διεργασίες, σήματα, επικοινωνίες, νήματα κλπ)
- ✧ Άλλα προχωρημένα θέματα και τεχνικές

Π2: Περίπου 50% της ύλης

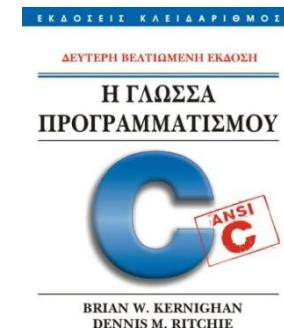
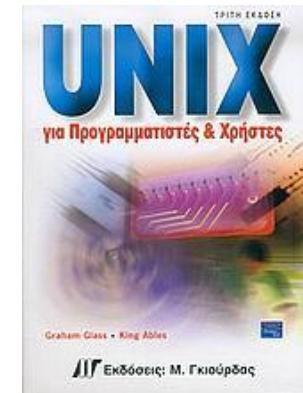
Συγγράμματα

- ❖ Υπάρχουν πολλά βιβλία για C
 - Και πάρα πολύ υλικό στο διαδίκτυο
- ❖ Για προγραμματισμό συστημάτων (POSIX) όχι τόσα πολλά μεταφρασμένα
 - Συνήθως θεωρούν δεδομένη τη γνώση της C
 - Πολλά που είναι για χρήση / διαχείριση του UNIX, όχι προγραμματισμό
 - ✧ Δεν μας αφορούν



Εύδοξος

- ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΕ UNIX,
M.J. Rochkind (2007)
 - ❖ Εξαιρετικό βιβλίο για προγραμματισμό συστημάτων UNIX
 - ❖ Υποθέτει γνώση της C
- UNIX ΓΙΑ ΠΡΟΓΡΑΜΜΑΤΙΣΤΕΣ ΚΑΙ ΧΡΗΣΤΕΣ,
G.Glass, K. Ables (2005)
 - ❖ Χρήση, διαχείριση, εσωτερικά του UNIX
 - ❖ Δύο κεφάλαια αφιερώνονται στα εργαλεία προγραμματισμού και στις κλήσεις συστήματος του UNIX και υποθέτει γνώση της C
- Σ ΑΠΟ ΤΗ ΘΕΩΡΙΑ ΣΤΗΝ ΠΡΑΞΗ,
Γ. Τσελίκης, Ν. Τσελίκης (2016)
 - ❖ Πάρα πολύ καλό βιβλίο C (ελληνικό!)
- Η ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ C,
B.W. Kernighan, D.M. Ritchie (2008)
 - ❖ «Ευαγγέλιο» της C (C90)
 - ❖ Ίσως όχι το καλύτερο για εκμάθηση.



- ❖ Πολλές ιστοσελίδες για C
- ❖ Πολλές ιστοσελίδες για προγραμματισμό σε UNIX/Linux/POSIX κλπ.
- ❖ Πολύ καλό βιόήθημα στο διάβασμά σας:
 - “Programming in C; Unix System Calls and Subroutines using C.”, A. D. Marshall
 - <http://www.cs.cf.ac.uk/Dave/C/CE.html>
 - Καλύπτει και εκμάθηση της C αλλά και αρκετό προγραμματισμό συστημάτων POSIX

Ώρες μαθήματος

❖ Διαλέξεις:

- Δευτέρα: 12:00 – 14:00
- Τετάρτη: 12:00 – 14:00 (*)
- Αίθουσα: I5

❖ Εργαστήρια:

- Τρίτη: 14:00 – 18:00 (20:00)
- ΠΕΠ I, ΠΕΠ II, ΠΕΛΣ

❖ Ήρες γραφείου διδάσκοντα:

- Τρίτη: 09:00 – 11:00
- B33

	Δε	Τρ	Τε	Πε	Πα
08:00					
09:00					
10:00		Γρ			
11:00					
12:00	Δ			Δ	
13:00					
14:00		Εργ			
15:00		Εργ			
16:00		Εργ			
17:00		Εργ			
18:00		Εργ			
19:00		Εργ			

Ιστοσελίδα μαθήματος:

<http://www.cse.uoi.gr/~dimako/teaching/>

Εργαστήρια (αν εγγράφεστε **ΠΡΩΤΗ ΦΟΡΑ ΣΤΟ ΜΑΘΗΜΑ**)

- ❖ Σκοπός/λειτουργία εργαστηρίων:
 - «Φροντιστηριακού» τύπου για ενίσχυση διδασκαλίας
 - Σχεδιασμός προγράμματος και υλοποίηση «επί τόπου»
 - (Πολύ) έμπειροι μεταπτυχιακοί και προσωπικό για να σας βοηθήσουν
- ❖ Οργανωτικά:
 - Ξεκινούν (λογικά) σε 2 εβδομάδες – Θα σας ενημερώσω εγκαίρως
 - Θα εργάζεστε δύο σε κάθε θέση (λόγω χώρου)
 - 14:00 – 18:00 (2 σειρές των 2 ωρών)
 - Σχεδόν κάθε εβδομάδα
 - **Παρουσίες**
- ❖ Βαθμός εργαστηρίου:
 - Θα βγει από τη συμμετοχή σας και από 2 προόδους
 - **20% οι παρουσίες + 40% πρόοδος1 + 40% πρόοδος2**
 - Κάθε απουσία σας αφαιρεί 10%. Από 2 και μετά δεν έχει διαφορά...

Όσοι ΠΕΡΑΣΟΥΝ το
εργαστήριο, το
κατοχυρώνουν για
πάντα

Και έρχονται μόνο στις
τελικές εξετάσεις.

Τι σημαίνει «ΠΕΡΝΑΩ» το εργαστήριο

- ❖ Για να **ΠΕΡΑΣΕΙ** κάποιος το εργαστήριο, θα πρέπει ο βαθμός του (20% οι παρουσίες + 40% πρόοδος 1 + 40% πρόοδος 2) να είναι $\geq 4,5$.
 - Όποιος το περάσει, το κατοχυρώνει για **PANTA**.

FAQ

- ❖ **ΕΡΩΤΗΣΗ:** Αν κάνω όλες τις παρουσίες περνάω;
- ❖ **ΑΠΑΝΤΗΣΗ:** ΟΧΙ, πρέπει να περάσετε και τις προόδους

- ❖ **ΕΡΩΤΗΣΗ:** Αν ΔΕΝ περάσω για κάποιο λόγο, μπορώ να ξαναπαρακολουθήσω το εργαστήριο του χρόνου;
- ❖ **ΑΠΑΝΤΗΣΗ:** ΟΧΙ, δείτε την επόμενη διαφάνεια

- ❖ **ΕΡΩΤΗΣΗ:** Δεν με βολεύει η ημ/νια της προόδου / έχω κανονίσει να λείπω / έχω πληρώσει εισιτήρια / θα είμαι άρρωστος εκείνη την ημέρα. Μπορώ να δώσω κάποια άλλη μέρα;
- ❖ **ΑΠΑΝΤΗΣΗ:** ΟΧΙ

Εργαστήρια (για όσους είχαν ΕΓΓΡΑΦΕΙ ΠΑΛΑΙΟΤΕΡΑ)

❖ Αφορά μόνο όσους:

1. Έχουν ξαναεγγραφεί στο μάθημα κατά το παρελθόν KAI
2. Δεν έχουν περάσει το εργαστήριο ποτέ

❖ Εγγραφή στα εργαστήρια:

- Υποχρεωτική, πλήρης (ηλεκτρονική) εγγραφή όπως όλοι
- **Αλλιώς ΔΕΝ ΘΑ ΜΠΟΡΕΣΟΥΝ ΝΑ ΕΞΕΤΑΣΤΟΥΝ λόγω χώρου**

❖ Εξέταση και βαθμός εργαστηρίων:

- Εξέταση στις 2 προόδους του εργαστηρίου, όπως όλοι.
- Απαραίτητη επικοινωνία με διδάσκοντα πριν από κάθε πρόοδο
- Ο βαθμός θα βγει ως εξής:
50% πρόοδος1 + 50% πρόοδος2

Εργαστήρια (για όσους είχαν ΕΓΓΡΑΦΕΙ ΠΑΛΑΙΟΤΕΡΑ)

- ❖ Για να **ΠΕΡΑΣΕΙ** κάποιος παλιός το εργαστήριο, θα πρέπει ο βαθμός του ($50\% \text{ πρόοδος}1 + 50\% \text{ πρόοδος}2$) να είναι $\geq 4,5$.

FAQ

- ❖ ΕΡΩΤΗΣΗ: *Είχα κάνει παρουσίες παλιά. Μπορώ να πάρω το 20% από αυτές;*
❖ ΑΠΑΝΤΗΣΗ: **ΟΧΙ**, οι παρουσίες μετρούν μόνο την 1^η φορά που εγγράφεστε στο μάθημα
- ❖ ΕΡΩΤΗΣΗ: *Μπορώ να ξαναπαρακολουθήσω το εργαστήριο να πάρω το 20% από τις παρουσίες;*
❖ ΑΠΑΝΤΗΣΗ: **ΟΧΙ**
- ❖ ΕΡΩΤΗΣΗ: *Μπορώ να ξαναπαρακολουθήσω το εργαστήριο ΧΩΡΙΣ να πάρω το 20% από τις παρουσίες;*
❖ ΑΠΑΝΤΗΣΗ: Πολύ δύσκολο, και μόνο αν υπάρχουν ελεύθερες θέσεις. Αν όμως θέλετε να ασχοληθείτε μόνοι σας θα υπάρχουν και οι ασκήσεις και οι απαντήσεις και οι βοηθοί.

Βαθμολόγηση

❖ Επιτυχία στο μάθημα προϋποθέτει:

1. Επιτυχία στο εργαστήριο (Βαθμός εργαστηρίων $\geq 4,5$)
 - ✧ Όσοι δεν επιτύχουν στο εργαστήριο, δεν έχουν δικαίωμα εξετάσεων
2. Τουλάχιστον βαθμό 4,5 στις εξετάσεις
 - ✧ Όσοι έχουν βαθμό < 4 στις εξετάσεις, δεν περνούν ακόμα και άριστα να πήγαν στο εργαστήριο.

❖ Τελικός βαθμός (το πιθανότερο):

- **40% εργαστήριο + 60% τελικές εξετάσεις**

Το σημερινό μάθημα

Εισαγωγικά στοιχεία για τη C



MYY502

❖ D. Ritchie, Bell Labs, 1972

- Με βάση προηγούμενη γλώσσα (B)
- Χρησιμοποιήθηκε για την υλοποίηση του λειτουργικού συστήματος UNIX
- Ευρεία διάδοση από τότε.

❖ Από αυτήν προέκυψαν / επηρεάστηκαν οι περισσότερες από τις πιο δημοφιλείς γλώσσες:

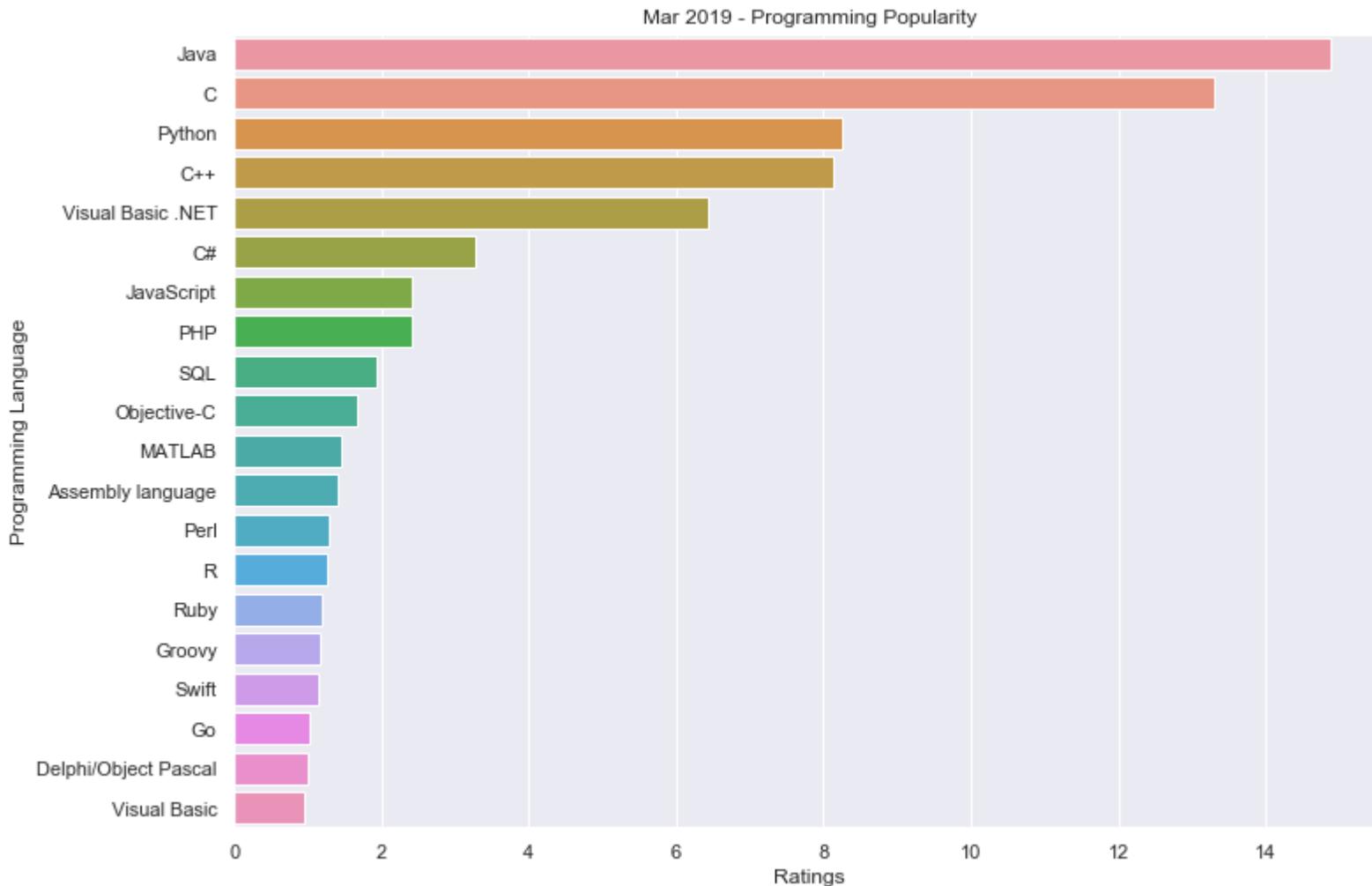
- Π.χ. C++, Java, PHP κλπ.

❖ Σε κάποιους τομείς (π.χ. ενσωματωμένα συστήματα) η C είναι ουσιαστικά η μοναδική επιλογή

- Δυνατή, μικρή, εύκολα μεταφράσιμη γλώσσα

Δημοτικότητα της C (TIOBE index, 2019)

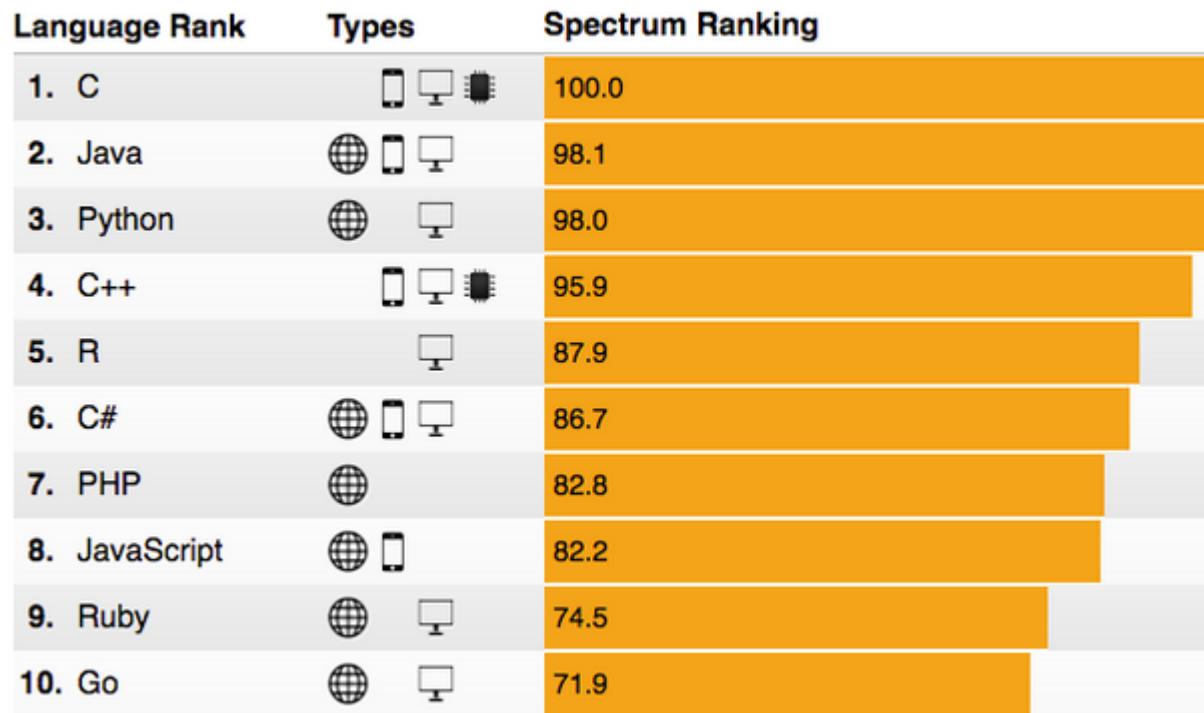
❖ Όλες οι κορυφαίες με βάση την C!



Κατάταξη γλωσσών προγραμματισμού (IEEE Spectrum)

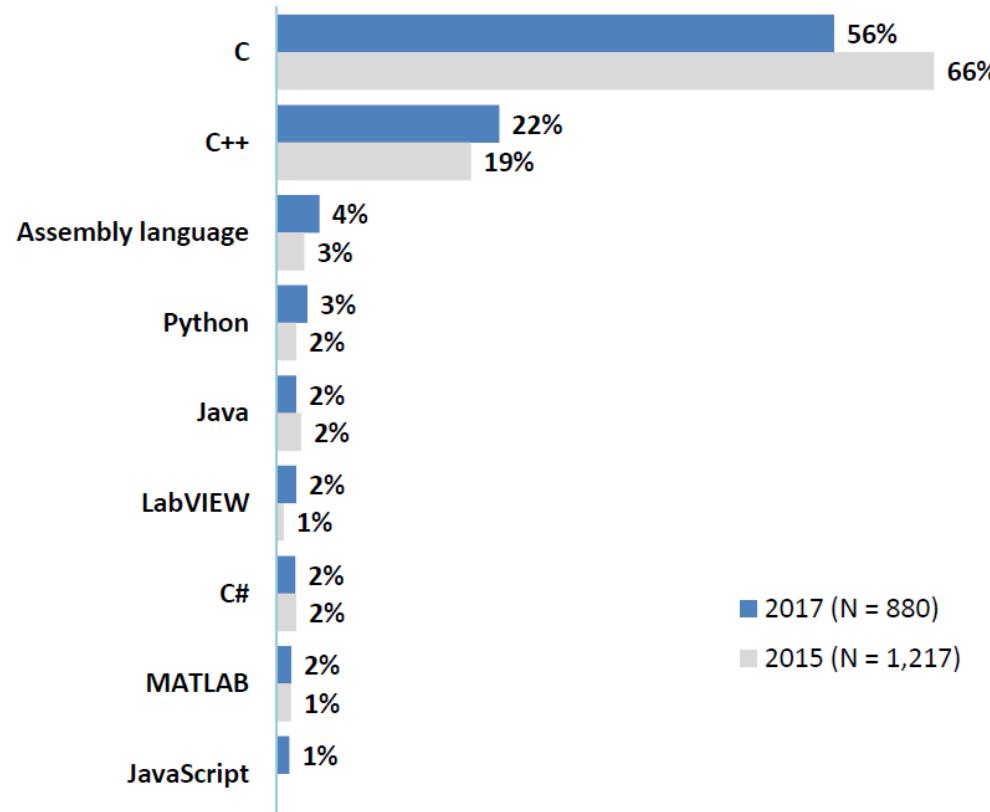
❖ IEEE, Αυγ. 2016

<http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages>



Στα ενσωματωμένα συστήματα

My current embedded project is programmed mostly in:



EE Times embedded

2017 Embedded Markets Study

© 2017 Copyright by AspenCore. All rights reserved.



- ❖ Το UNIX γράφτηκε σε C
 - ❖ Ο πυρήνας του Linux είναι γραμμένος σε C
 - ❖ Σχεδόν όλες οι εφαρμογές συστήματος είναι σε C
 - ❖ Η πλειονότητα εφαρμογών ανοιχτού κώδικα είναι σε C
 - ❖ ...
-
- ❖ **Προσοχή:** Η C ΔΕΝ είναι πανάκεια...
 - «Επικίνδυνη» αν κάποιος δεν την ξέρει καλά
 - «Χαμηλότερου» επιπέδου από άλλες γλώσσες (π.χ. Java)
 - Δεν βολεύει πάντα για εφαρμογές χρήστη, ειδικά όταν υπάρχει γραφική αλληλεπίδραση

Εισαγωγή στη C

C για προγραμματιστές Java



MYY502

Hello world

```
public class hello          #include <stdio.h>
{
    public static         int main() {
        void main (String args []) {     puts("Hello world");
            System.out.println      return 0;
            ("Hello world");           }
        }
}
```



❖ Κλάσεις

- Μόνο δεδομένα (μεταβλητές) και συναρτήσεις
- Η συνάρτηση `main()` είναι αυτή που εκτελείται αρχικά

❖ Boolean

- Με ακεραίους «προσομοιώνουμε» τα boolean
- Το **0** θεωρείται FALSE
- Οτιδήποτε μη-μηδενικό θεωρείται TRUE

❖ Strings (τουλάχιστον όπως τα χειρίζεται η Java)

- Χειρισμός μέσω πινάκων και δεικτών

❖ try ... catch μπλοκ (exceptions)

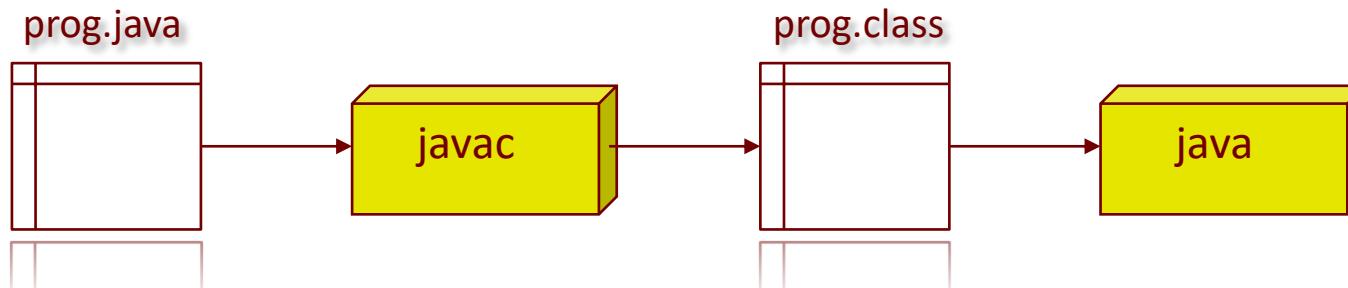
- Δεν υπάρχει ανάλογο, μόνο μέσω συναρτήσεων συστήματος

- ❖ Pointers (δείκτες)!
 - Όχι μόνο απλό πέρασμα με αναφορά
- ❖ «Ελευθερία» στους τύπους των δεδομένων (π.χ. int/short/char είναι πάνω-κάτω ίδιοι) και δεν γίνεται πλήρης έλεγχος κατά τη χρήση τους.
- ❖ «Ελευθερία» στη διαχείριση της μνήμης
 - Επαφίεται πλήρως στον προγραμματιστή
 - Η java έχει garbage collector που αυτόματα αποδεσμεύει άχρηστη μνήμη

Java vs C

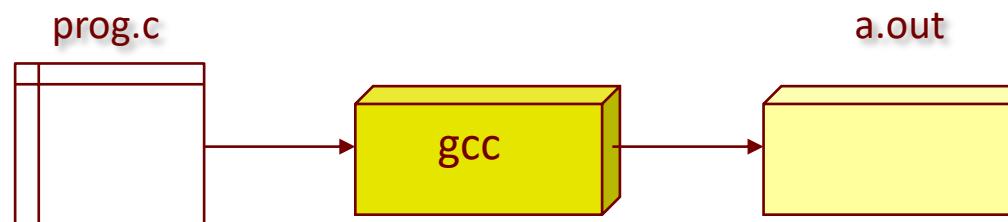
❖ Η Java είναι (βασικά) ερμηνευόμενη (*interpreted*)

- Συνήθως μετατρέπεται (javac) σε bytecode,
 - ✧ ο οποίος ερμηνεύεται από μία εικονική μηχανή (JVM),
 - η οποία εκτελείται (java) στην πραγματική μηχανή



❖ Η C είναι (βασικά) μεταφραζόμενη (*compiled*)

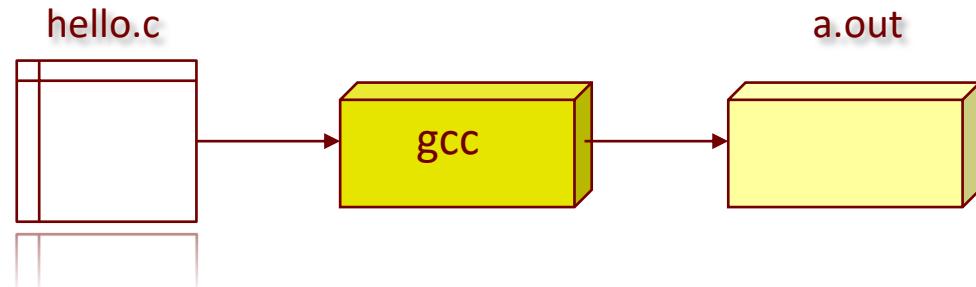
- Μετατρέπεται απευθείας σε εντολές assembly της πραγματικής μηχανής που θα εκτελέσει το πρόγραμμα
 - ✧ Το πρόγραμμα εκτελείται αυτόνομα



Το πρώτο πρόγραμμα σε C (hello.c)

```
#include <stdio.h>           ← HEADER (αρχείο επικεφαλίδων)  
                                Θυμίζει το import της java  
  
int main()                   ← Συνάρτηση εκκίνησης  
{  
    /* Just show a simple message */   ← Σχόλιο  
    printf("Hello, World\n");          ← Οθόνη  
}  
                                ← Τερματισμός προγράμματος  
                                ← Αλλαγή γραμμής
```

Μετάφραση του προγράμματος



- ❖ Στο τερματικό:

```
% ls
```

```
hello.c
```

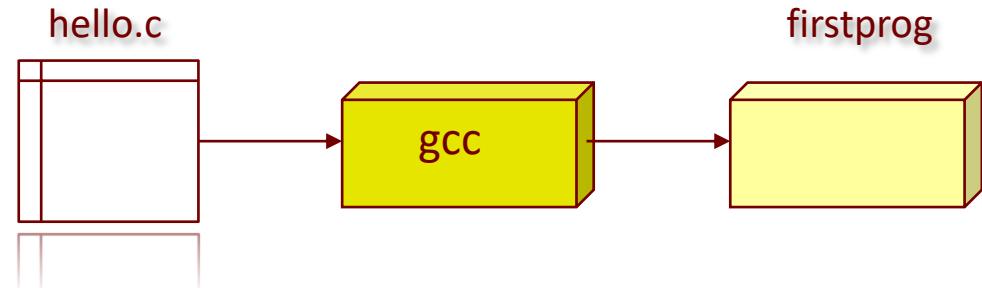
```
% gcc hello.c
```

```
% ls
```

```
a.out hello.c
```

- ❖ Ο μεταφραστής (gcc) ονομάζει το εκτελέσιμο “a.out”

Μετάφραση του προγράμματος με δικό μας όνομα

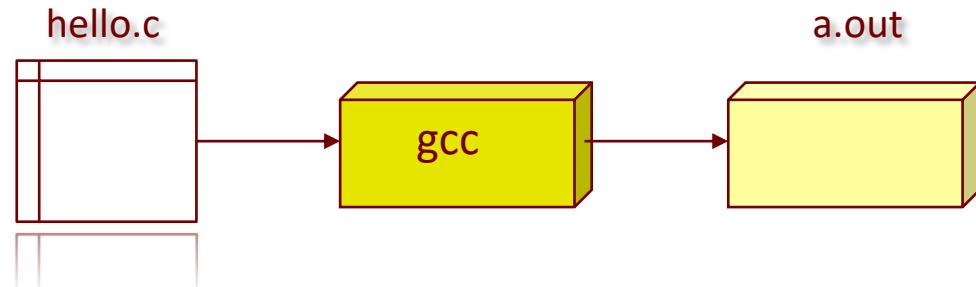


- ❖ Στο τερματικό:

```
% ls  
hello.c  
% gcc -o firstprog hello.c  
% ls  
firstprog hello.c
```

- ❖ Με το “–o” ο μεταφραστής αντί για “a.out” ονομάζει το εκτελέσιμο με ότι όνομα μας αρέσει.

Εκτέλεση του προγράμματος



- ❖ Στο τερματικό:

```
% ls
```

```
hello.c
```

```
% gcc hello.c
```

```
% ls
```

```
a.out hello.c
```

```
% ./a.out
```

```
Hello, World
```

```
%
```

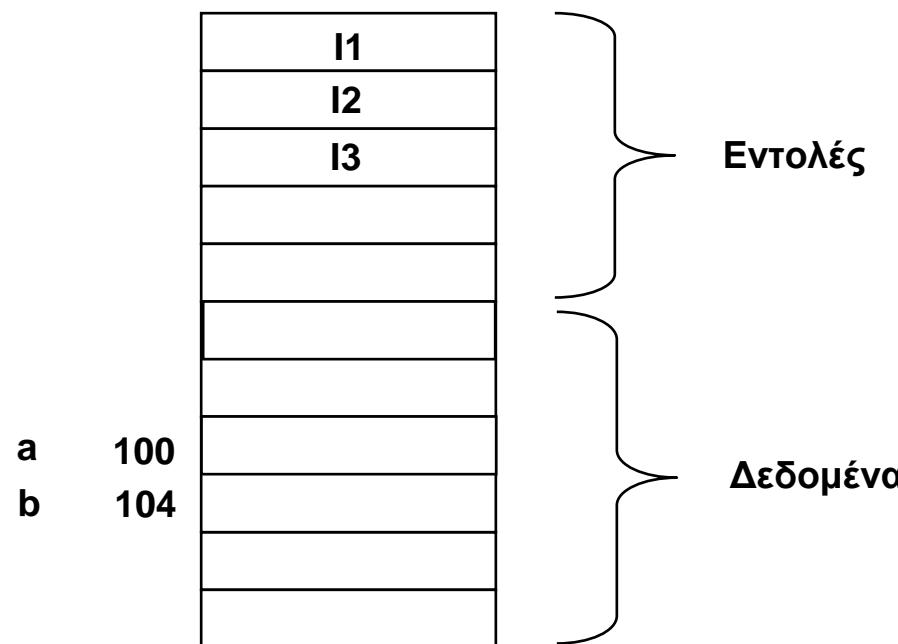
- ❖ Το “.” εννοεί το τρέχον directory – ίσως και να μην χρειάζεται.

- ❖ Πρόγραμμα = {δεδομένα} + {κώδικας (συναρτήσεις)}
- ❖ Συναρτήσεις = {main, ...}
- ❖ Δεδομένα = {μεταβλητές, σταθερές}
- ❖ Σταθερές = ποσότητες που δεν μεταβάλλονται κατά την εκτέλεση του προγράμματος
 - Π.χ. $\pi = 3.14$
- ❖ Μεταβλητές = ποσότητες που μεταβάλλονται
 - I/O, υπολογισμοί

- ❖ Τα δεδομένα αποθηκεύονται στη μνήμη του υπολογιστή
- ❖ Δηλώνοντας μια μεταβλητή δεσμεύω θέσεις στη μνήμη και καθορίζω ένα όνομα που χρησιμοποιώ για να αναφερθώ σε αυτή τη θέση μνήμης
- ❖ π.χ.
 - int const n = 10;
 - int a;

Εισαγωγικά

- ❖ Και οι εντολές αποθηκεύονται στη μνήμη του υπολογιστή
- ❖ Π.χ. εντολή I1: πρόσθεσε την τιμή της μεταβλητής a με αυτή της b → πρόσθεσε τα περιεχόμενα της θέσης 100 στα περιεχόμενα της θέσης 104



Μεταβλητές & τύποι δεδομένων

- ❖ Κάθε μεταβλητή, σταθερά έχει ένα τύπο
- ❖ Ο τύπος καθορίζει το μέγεθος του «κελιού» που θα δεσμευτεί στη μνήμη
 - char: 1 byte
 - int: 4 bytes (συνήθως)
 - float: 4 bytes
 - double: 8 bytes
- ❖ Προσδιοριστές
 - short int: 2 bytes
 - long int: 8 bytes
 - long double: ? bytes (≥ 10)

❖ Προσδιοριστές: `unsigned`

- Χωρίς πρόσημο (μόνο θετικοί) – όλα τα bits για την τιμή του αριθμού

❖ `unsigned int`, `unsigned short int`, `unsigned char`

- 32 bits, 16 bits, 8 bits

- ✧ 0, 1, ..., 4294967295
- ✧ 0, 1, ..., 65535
- ✧ 0, 1, ..., 255

❖ `int`, `short int`, `char`

- 31 bits, 15 bits, 7 bits (συν 1 για το πρόσημο)

- ✧ -2147483648, ..., 2147483647
- ✧ -32768, ..., 32767
- ✧ -128, ..., 127

Εκτυπώνοντας τις τιμές των μεταβλητών

- ❖ Η βασικότερη και γενικότερη συνάρτηση εκτύπωσης είναι η “printf”.
 - Αρχικά πρέπει να δώσω ως πρώτο όρισμα μια συμβολοσειρά με το ΤΙ ΤΥΠΟ θα εκτυπώσει
 - Στη συνέχεια, τα επόμενα ορίσματα είναι οι αντίστοιχες μεταβλητές ή εκφράσεις
- ❖ Παράδειγμα:

```
int main() {  
    int x = 5;  
    float y = 1.2;  
  
    printf("%d", x); /* %d ή %i για ακεραίους */  
    printf("%f", y); /* %f για πραγματικούς */  
    printf("x = %d and y = %f", x, y);  
    return 0;  
}
```

- ❖ Όσες μεταβλητές βρίσκονται εντός ενός μπλοκ εντολών (π.χ. μέσα σε μία συνάρτηση) είναι **τοπικές (local)** και μπορούν να χρησιμοποιηθούν μόνο εντός του μπλοκ.
- ❖ Όσες είναι εκτός των συναρτήσεων είναι **καθολικές (global)** και μπορούν να χρησιμοποιηθούν παντού.
- ❖ Περισσότερα αργότερα...

- ❖ Όλα τα δεδομένα σε ένα υπολογιστή κωδικοποιούνται ως ακολουθίες 0, 1
- ❖ Ένας χαρακτήρας κωδικοποιείται ως ακολουθία 0, 1
- ❖ Άρα στην ουσία ο υπολογιστής τον αντιλαμβάνεται σαν ένα αριθμό
 - Ο χαρακτήρας '0' αντιστοιχεί στον αριθμό 48
 - `char ch = 'x';`
 - Λέγοντας `ch = 'x'` είναι σαν να λέμε: βάλε στη μεταβλητή `ch` την τιμή (αριθμό) που αντιστοιχεί στο χαρακτήρα 'x'
 - `ch++;`
- ❖ Αριθμητική τιμή = κωδικός ASCII

Παράδειγμα με printf

```
#include <stdio.h>

int main()
{
    char ch = 'x';      /* Τοπική μεταβλητή τύπου χαρακτήρα */

    printf("ch = %d, ch = %c\n", ch, ch);

    return 0;
}
```

Ακέραιοι σε διάφορες μορφές...

```
int main() {
    int x = 95; /* 000...0 0101 1111 */

    printf("%d", x); /* 95 */
    printf("%x", x); /* 5f */
    printf("%o", x); /* 137 */
    printf("%c", x); /* _ */

    x = 95; /* Καταχωρώντας το 95 σε διάφορες μορφές */
    x = 0x5F;
    x = 0137;

    x = 'd'; /* Ποιος αριθμός είναι αυτός; */
    printf("%d", x); /* 100 */
    printf("%x", x); /* 64 */
    printf("%o", x); /* 144 */
    printf("%c", x); /* d */
    return 0;
}
```

- ❖ Αριθμητικοί τελεστές
 - +, -, *, /, % (το τελευταίο μόνο για ακεραίους)
- ❖ Συγκριτικοί τελεστές
 - >, <, <=, >=, ==, !=
- ❖ Λογικοί τελεστές
 - &&, ||, !
- ❖ Υπάρχουν και κάποιοι άλλοι τελεστές που θα μας απασχολήσουν αργότερα
 - Bitwise operators: ~, &, |, ^, >>, <<

Πράξεις και μετατροπές

- ❖ Αριθμητικοί τελεστές για δεδομένα ίδιου τύπου κυρίως (π.χ. πρόσθεση δύο ακεραίων)

- ❖ Όμως, μπορούμε να κάνουμε και πράξεις με μεταβλητές διαφορετικού τύπου, π.χ.

```
int x; float f;  
f = f+x;
```

- Γίνεται εσωτερική μετατροπή των «κατώτερων» τύπων σε «ανώτερους»
 - Και το αποτέλεσμα ανώτερου τύπου

- ❖ Τέτοιες μετατροπές γίνονται αυτόματα αλλά μπορούμε να τις ζητήσουμε και εμείς σε ένα πρόγραμμα με το μηχανισμό των “type casts”

- `f + x` (το `x` μετατρέπεται αυτόματα σε `float`)
 - `f + ((float) x)` (cast του προγραμματιστή)

❖ Τελεστές σύντμησης:

`++, --, +=, -=, *=, /=`

❖ Παράδειγμα

- `++i` και `i++` (pre-increment, post-increment)
- `i=i+1` και `i+=1`

❖ Παράδειγμα

`i = 3;`

`x = ++i; /* πρώτα γίνεται η αύξηση και μετά η αποτίμηση της έκφρασης */`

`x = i++; /* πρώτα γίνεται η αποτίμηση της έκφρασης και μετά η αύξηση */`

`i = i++ + ++i; /* εδώ τι τιμή θα πάρει τελικά το i? */`

Ο «τριαδικός» τελεστής

μεταβλητή = συνθήκη ? τιμή1 : τιμή2;

❖ Η εκτέλεση ισοδυναμεί με:

```
if (συνθήκη)
    μεταβλητή = τιμή1;
else
    μεταβλητή = τιμή2;
```

❖ Παράδειγμα:

```
x = (y > 0) ? 1 : 0;
```

2 «στυλ» σταθερών

❖ Στη java το “final” μπορεί να χρησιμοποιηθεί για να ορίσει «σταθερές»

❖ Στη C υπάρχουν 2 τρόποι να οριστούν σταθερές:

1. Με προσθήκη του «const» στον τύπο της δήλωσης, π.χ.

```
const int x = 5; /* Δεν μπορεί να αλλάξει τιμή */
```

2. Με ορισμό σταθεράς προεπεξεργαστή (#define)

```
#define M 10           /* Το σύνηθες ... */
```

```
#define PI 3.14
```

```
#define NEWLINE '\n'
```

❖ Διαφορά:

- Οι const καταλαμβάνουν μνήμη για αποθήκευση
- Οι #define ΑΝΤΙΚΑΘΙΣΤΑΝΤΑΙ ΠΡΙΝ ΓΙΝΕΙ Η ΜΕΤΑΦΡΑΣΗ του προγράμματος (και άρα δεν υπάρχουν στο εκτελέσιμο)

Διάβασμα (scanf) / εκτύπωση (printf)

```
#include <stdio.h> /* Απαραίτητο */

char c;                  /* Καθολική μεταβλητή */

int main() {             /* Συνάρτηση main */
    int i;                /* Τοπικές δηλώσεις ΠΑΝΤΑ στην αρχή της συνάρτησης */
    float f;

    printf("Dwse 1 xaraktira, 1 akeraio kai enan pragmatiko\n");
    scanf("%c", &c);
    scanf("%d%f", &i, &f);           /* Η scanf ΘΕΛΕΙ & στις μεταβλητές */
    printf("c = %c, i = %d, f = %f\n", c, i, f);
    return 0;
}
```

Εισαγωγή στη C

Εντολές



MYY502

❖ Όπως και στην java:

- if
- if-else
- switch
- for
- while
- do-while
- break

❖ Επιπλέον:

- goto

if & if-else

```
if (condition) {  
    statements;  
}  
  
if (condition) {  
    statements;  
} else {  
    statements;  
}
```

Αν υπάρχει ΜΟΝΟ ΈΝΑ statement,
όπως και στην Java, δε χρειάζονται οι
αγκύλες. Π.χ. ο παρακάτω κώδικας

```
if (x == 3)  
    x++;  
    y++;  
z = x+y;
```

είναι ισοδύναμος με:

```
if (x == 3) {  
    x++;  
}  
y++;  
z = x+y;
```

switch

```
switch (variable) {  
    case const1:  
        statements;  
        break;  
    case const2:  
        statements;  
        break;  
    ...  
    default:  
        statements;  
        break;  
}
```

Εντολές ελέγχου – switch

```
switch (variable) {  
    case const1:  
        statements;  
        break;  
    case const2:  
        statements;  
        break;  
    ...  
    default:  
        statements;  
        break;  
}
```

```
switch ( choice )  
{  
    case 'a':  
    case 'A':  
        do_thing_1();  
        break;  
  
    case 'b':  
    case 'B':  
        do_thing_2();  
        break;  
    ...  
    default:  
        printf("Wrong choice.");  
}
```

❖ Προσοχή:

- Αν δεν υπάρχει break, η εκτέλεση ενός case συνεχίζεται με τον κώδικα του επόμενου case...

while & do-while

```
while (condition) {  
    statements;  
}
```

```
x = 0;  
while (x < 10)  
    x++;
```

```
x = 0;  
while (x < 10);  
    x++;
```

```
do {  
    statements;  
} while (condition);
```

Αν υπάρχει ΜΟΝΟ ΈΝΑ statement,
όπως και στη Java, δε χρειάζονται οι
αγκύλες.

for (I)

```
for (initialization; condition; iteration) {  
    statements;  
}
```

```
/* Ισοδύναμος κώδικας: */  
initialization;  
while (condition) {  
    statements;  
    iteration;  
}
```

Αν υπάρχει ΜΟΝΟ ΕΝΑ statement,
όπως και στην Java, δε χρειάζονται οι
αγκύλες.

- ❖ Τα initialization / iteration μπορούν να περιέχουν πολλές εκφράσεις, χωρισμένες με κόμμα.

for (II)

- ❖ Τα initialization / iteration μπορούν να περιέχουν πολλές εκφράσεις, χωρισμένες με κόμμα.

Π.χ.

```
int i, sum;  
sum = 0;  
for (i = 1; i <= 10; i++) {  
    sum += i;  
}
```

```
for (i = 1, sum = 0; i <= 10; sum += (i++))  
    ;
```

for (III)

- ❖ Δεν υπάρχει το for each

~~for (δήλωση : συλλογή)~~

της Java.

break

- ❖ Έξοδος από switch ή από βρόχους for/while/do, π.χ.

```
while (1) {  
    if (w == 3) {  
        break; /* Βγαίνει αμέσως μετά το while */  
    }  
    . . .  
}
```

- ❖ Δεν υπάρχει ονοματισμένο break (τύπου break <label>)

goto

- ❖ Η εκτέλεση μεταπηδά σε συγκεκριμένη ετικέτα, π.χ.

```
if (x == 1) {  
    goto before;  
}  
y = 2;  
goto after;  
before: y = 1;  
after: ...
```

- ❖ Επικίνδυνη / μη-προβλέψιμη εντολή, π.χ.

```
while (1) {  
    if (w == 3) {  
        Strange: x=1; ...  
    }  
    ...  
    if (condition) goto Strange;
```

- ❖ «Ακατάλληλη» δια ανηλίκους. Κακή προγραμματιστική τεχνική. Δεν πρέπει να χρησιμοποιείται σχεδόν ποτέ.

Αγκύλες και μπλοκ κώδικα

- ❖ Γράψτε τον παρακάτω κώδικα ΧΩΡΙΣ αγκύλες, όπου γίνεται:

```
if (i >= 81) {  
    x = 3;  
}  
if (row >= 1 && col<= 9) {  
    q = 5;  
    y = x;  
}  
else {  
    for (S[row][col] = 1; S[row][col] <= 9; S[row][col]++) {  
        if ( sudoku_solve(i) ) {  
            return (1);  
        }  
    }  
}
```

Αγκύλες και μπλοκ κώδικα

❖ Πρώτη προσπάθεια:

```
if (i >= 81) {  
    x = 3;  
}  
if (row >= 1 && col<= 9) {  
    q = 5;  
    y = x;  
}  
else {  
    for (S[row][col] = 1; S[row][col] <= 9; S[row][col]++) {  
        if (sudoku_solve(i)) {  
            return (1);  
        }  
    }  
}
```

Αγκύλες και μπλοκ κώδικα

❖ Δεύτερη προσπάθεια:

```
if (i >= 81
    x = 3;

if (row >= 1 && col<= 9) {
    q = 5;
    y = x;
}
else {
    for (S[row][col] = 1; S[row][col] <= 9; S[row][col]++) {
        if ( sudoku_solve(i) )
            return (1);

    }
}
```

Αγκύλες και μπλοκ κώδικα

❖ Τρίτη προσπάθεια:

```
if (i >= 81
    x = 3;

if (row >= 1 && col<= 9) {
    q = 5;
    y = x;
}
else {
    for (S[row][col] = 1; S[row][col] <= 9; S[row][col]++)
        if ( sudoku_solve(i) )
            return (1);

}
```

Αγκύλες και μπλοκ κώδικα

❖ Τελικός κώδικας:

```
if (i >= 81
    x = 3;

if (row >= 1 && col<= 9) {
    q = 5;
    y = x;
}
else
    for (S[row][col] = 1; S[row][col] <= 9; S[row][col]++)
        if ( sudoku_solve(i) )
            return (1);
```

❖ ΠΑΝΤΑ ΝΑ ΒΑΖΕΤΕ ΑΓΚΥΛΕΣ, ΑΚΟΜΑ ΚΑΙ ΌΤΑΝ ΔΕΝ ΧΡΕΙΑΖΕΤΑΙ

ΜΥΥ502

Προγραμματισμός Συστημάτων



Β. Δημακόπουλος

dimako@cse.uoi.gr

<http://www.cse.uoi.gr/~dimako>

Εργαστήρια

- ❖ (Για τους «παλιούς» έχει βγει ήδη φόρμα επανεγγραφής)
- ❖ Ξεκινούν την επόμενη εβδομάδα
 - Τρίτη, 15/10/2019
- ❖ Εγγραφές στο εργαστήριο
 - 2 βάρδιες, 14:00 – 16:00 και 16:00 – 18:00
 - Εγγραφές **ΑΤΟΜΙΚΕΣ** από αύριο μέχρι την επόμενη Δευτέρα
 - ✧ Θα υπάρξει και ανακοίνωση στην ιστοσελίδα του μαθήματος
- ❖ Προσωπικό Εργαστηρίου:
 - Υπεύθυνη: Β. Σταμάτη (Ε.ΔΙ.Π. ΤΜΗΥΠ)
 - Βοηθοί: Η. Κασμερίδης, Η. Κλεφτάκης, Π. Δημητρακόπουλος (Μεταπτυχιακοί)
- ❖ Θα ανακοινωθούν και ώρες γραφείου των παραπάνω



Εισαγωγή στη Σ

Πίνακες



MYY502

❖ Διαφορές με java:

- Όταν τους ορίζεις, πρέπει να δηλώσεις και το μέγεθος
- Οι αγκύλες πάνε μετά το όνομα της μεταβλητής

❖ Δηλ. δεν παίζει το:

```
int [] myarray;  
myarray = new int[30];
```

❖ Σωστή (και μοναδική) δήλωση:

```
int myarray[30];
```

❖ Η αρίθμηση των στοιχείων ξεκινάει από τη θέση μηδέν (0)

Πίνακες

- ❖ ΔΕΝ μπορεί να αλλάξει το μέγεθος του πίνακα.
- ❖ ΔΕΝ θυμάται / ελέγχει η C τα όρια του πίνακα.
- ❖ ΔΕΝ μπορεί να γίνει αρχικοποίηση του πίνακα παρά μόνο τη στιγμή της δήλωσής του. Μετά καταχώρηση στοιχείο-στοιχείο. Παράδειγμα:

```
#include <stdio.h>
#define N 10

int main() {
    int myarray[N] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    myarray[3] = 400;
    myarray[10] = 900;      /* Εκτός ορίων – ΔΕΝ μας προειδοποιεί
                           με κάποιο σφάλμα η C */
    ...
}
```

Πίνακες χωρίς μέγεθος

- ❖ Επιτρέπεται να μην δοθεί το μέγεθος του πίνακα κατά τη δήλωση *μόνο αν προκύπτει από την αρχικοποίησή του:*

```
int main() {  
    int myarray[] = {10,20,30,40};  
    ...  
}
```

- Προκύπτει (και δεν μπορεί να αλλάξει) ότι το μέγεθος είναι 4 στοιχεία.
- ❖ Το παρακάτω δεν επιτρέπεται!

```
int main() {  
    int myarray[];  
    ...  
}
```

Πίνακες - αρχικοποίηση

- ❖ Αν ο πίνακας δεν αρχικοποιηθεί κατά τη δήλωσή του
 - τα στοιχεία του θα έχουν τυχαίες τιμές («σκουπίδια»).
- ❖ Αν κατά τη δήλωση του πίνακα αρχικοποιηθούν λιγότερα στοιχεία από αυτά που έχει,
 - τα υπόλοιπα αρχικοποιούνται αυτόματα στο 0 (μηδέν)
- ❖ Π.χ. εύκολος τρόπος να αρχικοποιήσω όλα τα στοιχεία ενός πίνακα στο 0:
`int myarray[100] = { 0 };`

Συμβολοσειρές (strings): βασικά πίνακες χαρακτήρων

- ❖ Παράδειγμα:

```
int main() {  
    char str1[5];  
    char str2[6] = {'b', 'i', 'l', 'l', '\0'};  
  
    str2[1] = 'a'; /* b a l l */  
    ...  
}
```

- ❖ Απλά πρέπει να υπάρχει στο τέλος και ο ειδικός χαρακτήρας '\0' (λεπτομέρειες αργότερα).
- ❖ Πολύ πιο εύκολη αρχικοποίηση, μόνο για συμβολοσειρές:

```
char str3[6] = "maria"; /* Υπονοείται το \0 στο τέλος */  
char str4[] = "demi"; /* Μέγεθος 5 */
```



Πίνακες 2D

- ❖ Για διδιάστατους (τριδιάστατους κλπ), κατά τη δήλωση χρησιμοποιούνται πολλαπλές αγκύλες για να δηλώσουν το μέγεθος κάθε διάστασης.

```
int array1[10][20];           /* 10 γραμμών, 20 στηλών */  
  
int main() {  
    int array2[30][20];         /* 30 γραμμών, 20 στηλών */  
  
    array1[0][0] = array2[10][10];  
    array1[5] = array2[3]; /* Δεν επιτρέπεται */  
}
```

- ❖ Οι πίνακες αυτοί είναι ΠΡΑΓΜΑΤΙΚΑ διδιάστατοι και ΣΤΑΘΕΡΟΙ
 - Όχι σαν τη java όπου βασικά πρόκειται για διάνυσμα όπου κάθε στοιχείο του είναι άλλο διάνυσμα (γραμμή).
 - Όλες οι γραμμές ίδιου (αμετάβλητου) μεγέθους (σε αντίθεση με τη java που κάθε γραμμή μπορούσε να έχει διαφορετικό μέγεθος).

Η γλώσσα C

Συναρτήσεις



- ❖ Πρόκειται απλά για μια ακολουθία εντολών την οποία χρησιμοποιούμε σε αρκετά σημεία του προγράμματός μας και την οποία την ξεχωρίζουμε ως «υποπρόγραμμα», ώστε να μπορεί να κληθεί και να εκτελεστεί όσες φορές θέλουμε.
- ❖ «Μέθοδοι» στη Java.
- ❖ Στη C δεν αποτελούν μέρος καμίας κλάσης (δεν υπάρχουν κλάσεις), καλούνται από παντού.



Τα 3 βασικά σημεία των συναρτήσεων: (α) πρωτότυπο

❖ Πρωτότυπο (prototype) – **function declaration**

- **δήλωση** (declaration) που περιγράφει τι χρειάζεται η συνάρτηση για να λειτουργήσει και τι είδους αποτέλεσμα θα επιστρέψει:

<τύπος_επιστροφής> όνομα_συνάρτησης (<τύπος_παραμέτρου> παράμετρος, ...) ;

Π.χ.

```
int power(int base, int exponent);
```

❖ Είναι απαραίτητο το πρωτότυπο;

- Απάντηση: όχι πάντα, αλλά αν δεν το γράψετε μπορεί να δημιουργηθούν προβλήματα.
- Θεωρείστε το ισοδύναμο με τη **δήλωση μίας μεταβλητής** (η οποία πρέπει να οριστεί / δηλωθεί πριν την χρησιμοποιήσετε)

Τα 3 βασικά σημεία των συναρτήσεων: (β) ορισμός

❖ Ορισμός / υλοποίηση της συνάρτησης – **function definition**

```
<πρωτότυπο>          /* Προσοχή: χωρίς το ';' */
{
    ...
    /* οι εντολές της συνάρτησης */
    return (<τιμή>);
}
```

❖ Η **return**: τερματίζει τη συνάρτηση και επιστρέφει το αποτέλεσμα.

- Τι σημαίνει επιστρέφει το αποτέλεσμα?
- Αν έχω στη **main()** : “εκτέλεσε τη συνάρτηση **power** για τα 2, 4” τότε ό,τι τιμή γίνεται **return** αποθηκεύεται σε μια προσωρινή θέση μνήμης **tmp**.
 - ✧ **res = power(2, 4)** → **res = tmp** όπου στο **tmp** έχει μπει το 2^4 , δηλ. το 16

❖ Ειδικός τύπος συνάρτησης: **void**

- Αν η συνάρτηση είναι τύπου **void** τότε ΔΕΝ επιστρέφει τίποτε
- Έχουμε σκέτο:
return;

❖ Κλήση μίας συνάρτησης – **function call**

- Μέσα στο πρόγραμμα
 - ✧ “εκτέλεσε τη συνάρτηση power για τα 2, 4 και υπολόγισε το αποτέλεσμα”
- Τρόπος κλήσης:
`<όνομα_συνάρτησης>(τιμές για πραγματικές παραμέτρους)`

Παράδειγμα

```
#include <stdio.h>

int main() {
    int x, y, k, n, res;
    int power(int base, int n); /* πρωτότυπο */

    x = 2; y=3; k=4; n=2;
    res = power(x, k); /* κλήση */
    printf("%d \n", res);
    res = power(y, n); /* κλήση */
    printf("%d \n", res);

    return 0; /* Η main() πρέπει να επιστρέψει 0 αν όλα OK */
}

/* Ορισμός (υπολογίζει το base υψωμένο στη δύναμη n, basen) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
        p = p*base;
    return p; /* τιμή επιστροφής */
}
```

Παράμετροι (parameters)
ή
τυπικές παράμετροι
(formal parameters)

Ορίσματα (arguments) ή
πραγματικές παράμετροι

Παράδειγμα – αλλού το πρωτότυπο

```
#include <stdio.h>
int power(int base, int n);           /* πρωτότυπο */

int main() {
    int x, y, k, n, res;

    x = 2; y=3; k=4; n=2;
    res = power(x, k);                /* κλήση */
    printf("%d \n", res);
    res = power(y, n);                /* κλήση */
    printf("%d \n", res);

    return 0;                         /* Η main() πρέπει να επιστρέψει 0 αν όλα OK */
}

/* Ορισμός (υπολογίζει το  $base^n$ ) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
        p = p*base;
    return p;                          /* τιμή επιστροφής */
}
```

ΔΙΑΦΟΡΑ ΜΕ ΠΡΙΝ:

Όχι μόνο η `main()`, αλλά ΟΛΕΣ οι επόμενες συναρτήσεις θα «γνωρίζουν» την `power()`.

Πριν, ΜΟΝΟ στη `main()` είχε γίνει γνωστή!

Παράδειγμα χωρίς πρωτότυπο. Τι θα γίνει;

```
#include <stdio.h>

int main() {
    int x, y, k, n, res;

    x = 2; y=3; k=4; n=2;
    res = power(x, k);           /* κλήση */
    printf("%d \n", res);
    res = power(y, n);           /* κλήση */
    printf("%d \n", res);

    return 0;                    /* Η main() πρέπει να επιστρέψει 0 αν όλα OK */
}

/* Ορισμός (υπολογίζει το basen) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
        p = p*base;
    return p;                   /* τιμή επιστροφής */
}
```

ΑΠΟΤΕΛΕΣΜΑ:

Compiler **warnings**: δεν «γνωρίζει» την `power()` στα σημεία που γίνονται οι δύο κλήσεις. Πολλές φορές υπάρχουν και καταστροφικά αποτελέσματα.

Παράδειγμα χωρίς πρωτότυπο. Τι θα γίνει;

```
#include <stdio.h>

/* Ορισμός (υπολογίζει το  $base^n$ ) */
int power(int base, int n) {
    int i, p;
    for (i = p = 1; i <= n; i++)
        p = p*base;
    return p;          /* τιμή επιστροφής */
}

int main() {
    int x, y, k, n, res;

    x = 2; y=3; k=4; n=2;
    res = power(x, k);           /* κλήση */
    printf("%d \n", res);
    res = power(y, n);           /* κλήση */
    printf("%d \n", res);

    return 0;                    /* Η main() πρέπει να επιστρέψει 0 αν όλα OK */
}
```

ΑΠΟΤΕΛΕΣΜΑ:

Μιας και η συνάρτηση ορίστηκε ΠΡΙΝ τα σημεία των δύο κλήσεων, ΘΕΩΡΕΙΤΑΙ ΓΝΩΣΤΗ.

Επομένως δεν δημιουργείται πρόβλημα και άρα δεν είναι απαραίτητο το πρωτότυπο.

ΜΗΝ ΤΟ ΚΑΝΕΤΕ!!

ΠΑΝΤΑ ΝΑ ΓΡΑΦΕΤΕ ΤΑ ΠΡΩΤΟΤΥΠΑ!!

```
int power(int base, int n) {  
    int i, p;  
    for (i = p = 1; i <= n; i++)  
        p = p*base;  
    return p;  
}
```

- ❖ Κανένα πρόβλημα;
 - Τι γίνεται αν $n < 0$?
- ❖ Πώς θα το áλλαζα για να ελέγχει αν το n είναι θετικός;

```
if (n < 0) {  
    return (-1);  
}
```

Έτοιμες μαθηματικές συναρτήσεις στη C

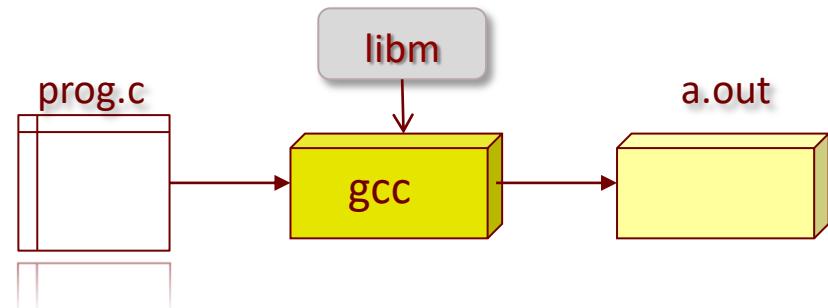
❖ Πρέπει:

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

```
int main()
{
    printf("cos(0) = %lf\n", cos(0.0));
    return 0;
}
```

❖ Επίσης, κατά τη μεταγλώττιση, πρέπει να συμπεριληφθεί και η βιβλιοθήκη των μαθηματικών συναρτήσεων με το **-lm**:

```
% gcc prog.c -lm
% ls
a.out prog.c
```



Συνήθεις μαθηματικές συναρτήσεις

```
double acos(double);  
double asin(double);  
double atan(double);  
double cos(double);  
double cosh(double);  
double sin(double);  
double sinh(double);  
double tan(double);  
double tanh(double);
```

```
double sqrt(double);  
double exp(double);  
double pow(double, double);  
double log(double);  
double log10(double);  
double fabs(double);
```

Πέρασμα Παραμέτρων

- ❖ Οι μεταβλητές βασικού τύπου (int, char, float, ...) περνιούνται ως παράμετροι σε μια συνάρτηση **με τιμή (by value)** .
 - δηλαδή αντιγράφονται οι τιμές τους σε τοπικές μεταβλητές της συνάρτησης.
 - όποιες αλλαγές γίνουν σε αυτές τις τιμές των μεταβλητών στο εσωτερικό της συνάρτησης δεν είναι εμφανείς στο κομμάτι του προγράμματος στο οποίο έγινε η κλήση της συνάρτησης.
- ❖ Αν θέλουμε να ξεπεράσουμε τον παραπάνω περιορισμό, δηλ. ότι αλλαγές γίνουν στις τιμές των μεταβλητών στο εσωτερικό της συνάρτησης να είναι εμφανείς στο κομμάτι του προγράμματος στο οποίο έγινε η κλήση της συνάρτησης, μπορούμε να περάσουμε τις διευθύνσεις των θέσεων μνήμης που έχουν δεσμευθεί για τις μεταβλητές, να κάνουμε όπως λέμε **πέρασμα με αναφορά (by reference, με χρήση παραμέτρων τύπου pointer)**.
- ❖ Οι παράμετροι τύπου **πίνακα** πάντα περνιούνται **με αναφορά**.
 - όποιες αλλαγές γίνουν στα στοιχεία του πίνακα είναι εμφανείς στο κομμάτι του προγράμματος στο οποίο έγινε η κλήση της συνάρτησης.

swap

```
#include <stdio.h>
void swap (int x, int y);

int main() {
    int a,b;
    a = 6;
    b = 7;
    swap(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
```

```
#include <stdio.h>
void swap (int *x1, int *x2);

int main() {
    int a,b;
    a = 6;
    b = 7;
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

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

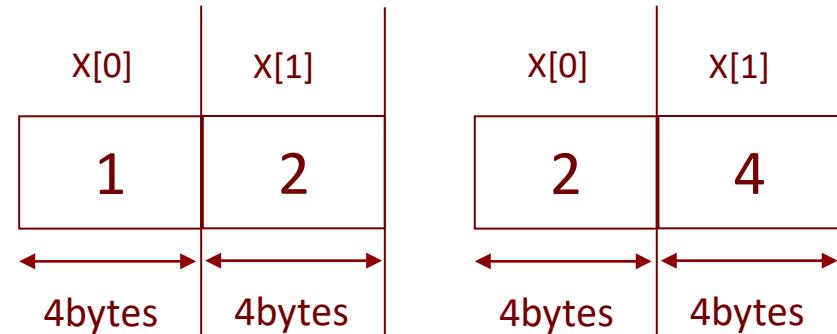
```
void swap (int *x1, int *x2) {
    int temp;
    temp = *x1;
    *x1 = *x2;
    *x2 = temp;
    return;
}
```

Παράδειγμα πέρασμα by reference με πίνακα

```
#include <stdio.h>
void f(int a[]);

int main() {
    int i;
    int x[] = {1, 2};
    f(x);
    for (i = 0; i < 2; i++)
        printf("%d\n", x[i]);
    return 0;
}

void f(int k[]) {
    int i;
    for (i = 0; i < 2; i++)
        k[i] = k[i]*2;
}
```



Στοίβα (stack)

- ❖ Για κάθε κλήση συνάρτησης έχω ένα κομμάτι μνήμης αφιερωμένο σε αυτή
- ❖ Δεσμεύεται αυτόματα κατά την είσοδο στη συνάρτηση και αποδεσμεύεται επίσης αυτόματα κατά την έξοδο από αυτή
- ❖ Ο χώρος μνήμης που χρησιμοποιείται για το σκοπό αυτό καλείται **stack** (στοίβα)
 - Το συγκεκριμένο κομμάτι μνήμης για μια συνάρτηση καλείται **stack frame**
- ❖ Στη μνήμη αυτή (**stack frame**) τοποθετούνται οι παράμετροι και οι τοπικές μεταβλητές μιας συνάρτησης

Η γλώσσα C

*Εμβέλεια και διάρκεια ζωής
μεταβλητών
(scope & lifetime)*



MYY502

Εμβέλεια μεταβλητών (scope / visibility)

- ❖ Τι είναι: Η **εμβέλεια** του ονόματος μιας μεταβλητής είναι το **τμήμα του προγράμματος στο οποίο η μεταβλητή μπορεί να χρησιμοποιηθεί**
 - Global (καθολική)
 - Local (τοπική, σε συνάρτηση)
 - Block (σε δομημένο μπλοκ εντολών { .. })
- ❖ Εμβέλεια τοπικών μεταβλητών → δηλώνονται στην αρχή κάθε συνάρτησης και μπορούν να χρησιμοποιηθούν μόνο μέσα στη συνάρτηση
- ❖ Εμβέλεια block → δηλώνεται στην αρχή ενός δομημένου block { ... } και είναι ορατό μόνο μέσα στο *block* αυτό!
- ❖ Εμβέλεια καθολικών μεταβλητών → από εκεί που δηλώθηκαν μέχρι τέλος αρχείου

Εμβέλεια

```
int i=2, j=7;

int f(int i) {
    return i+3;
}

int g() {
    int k;
    if (j > 5) {
        int j;
        j = f(i);
        k = 2*j;
    }
    return (k);
}

main() {
    i = g();
}
```

Η εμβέλεια προκύπτει ΜΟΝΟ από το κείμενο του κώδικα – απλά παρατηρώντας που είναι τοποθετημένες οι δηλώσεις σε σχέση με τις συναρτήσεις και τα blocks

KAI OXI

από το πως εκτελείται ο κώδικας.

Τι θα τυπωθεί;

```
#include <stdio.h>
char color = 'B';

void paintItGreen() {
    char color = 'G';
    printf("color@pIG: %c\n", color);
}

void paintItRed() {
    char color = 'R';
    printf("\n\t-----start pIR----\n\t");
    printf("color@pIR: %c\n\t", color);
    paintItGreen();
    printf("\tcolor@pIR: %c\n\t", color);
    printf("-----end pIR----\n\n");
}

main() {
    printf("\t\tcolor@main: %c\n", color);
    paintItGreen();
    printf("\t\tcolor@main: %c\n", color);
    paintItRed();
    printf("\t\tcolor@main: %c\n", color);
}
```

Διάρκεια (lifetime)

- ❖ Το **χρονικό διάστημα για το οποίο δεσμεύεται μνήμη για τις μεταβλητές**.
- ❖ **Τοπικές** μεταβλητές: ως την ολοκλήρωση της συνάρτησης στην οποία είναι ορισμένες
- ❖ **Καθολικές**: ως την ολοκλήρωση της εκτέλεσης του προγράμματος
- ❖ **Τοπικές** μεταβλητές:
 - **auto** (το default, υπάρχουν όσο διαρκεί το block / συνάρτηση, μιας και αποθηκεύονται συνήθως στη στοίβα)
 - **static** (διατηρείται ο χώρος ακόμα και αν τελειώσει μία συνάρτηση)

Διάρκεια – μνήμη

```
int i=2, j=7;
```

```
int f(int i) {  
    return i+3;  
}
```

```
int g() {  
    int k;  
    if (j > 5) {  
        int j;  
        j = f(i);  
        k = 2*j;  
    }  
    return (k);  
}
```

```
main() {  
    i = g();  
}
```

Η διάρκεια προκύπτει από την
εκτέλεση του κώδικα.

Μεταβλητές με διάρκεια static

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

```
void f(void) {  
    static int y = 0;  
    y++;  
    printf("%d\n", y);  
}
```

```
int main() {  
    int i;  
    for (i=0; i<5; i++)  
        f();  
    return 0;  
}
```

1. Η αρχικοποίηση γίνεται κατά την εκκίνηση του προγράμματος – δεν γίνεται στην κάθε κλήση της συνάρτησης.
2. Η μεταβλητή συνεχίζει να υπάρχει στη μνήμη ακόμα και μετά τη λήξη της συνάρτησης

Αρχικοποίηση μεταβλητών

❖ Με τη δήλωση:

- Οι καθολικές (global) και οι τοπικές static
 - ✧ Αρχικοποιούνται ΑΥΤΟΜΑΤΑ στο μηδέν (0)
- Οι τοπικές (εκτός των static)
 - ✧ Τυχαία αρχική τιμή (συνήθως σκουπίδια)

❖ Αρχικοποίηση πινάκων:

- `int a[] = {1, 2, ...};`
 - ✧ Από το πλήθος των στοιχείων προκύπτει το μέγεθος του πίνακα
- `int a[5] = {1, 2, ...};`
 - ✧ Αν παραπάνω από 5 στοιχεία στις αγκύλες → Error!
 - ✧ Αν λιγότερα από 5 → 0 στις υπόλοιπες θέσεις του a.

❖ Αρχικοποίηση πινάκων χαρακτήρων:

- `char txt[] = {'A', 'l', 'a', 'l', 'a', '\0'};`
- ή πιο εύκολα: `char txt[] = "Alala";`

Η γλώσσα C

*Εμβέλεια extern &
πολλαπλά αρχεία κώδικα*



MYY502

Πολλαπλά αρχεία κώδικα - scope.c

```
#include <stdio.h>
#define SCALE 2
int number;      /* Δήλωση/ορισμός καθολικής μεταβλητής */
int f(int param); /* Πρωτότυπο */

int main() {
    int y;
    number = 5;
    y= f(4*SCALE);
    printf("%d, %d\n", number, y);
    return 0;
}

int f(int x) {
    int y;
    y = x*SCALE + number;
    return (y);
}
```

Σε δύο αρχεία

scope.c

```
#include <stdio.h>
#define SCALE 2

int number, /* Δήλωση/ορισμός */
    f(int param); /* Πρωτότυπο */

int main() {
    int y;
    number = 5;
    y = f(4*SCALE);
    printf("%d,%d\n",number,y);
    return 0;
}
```

func.c

```
#define SCALE 2

/* Μεταβλητή ορισμένη αλλού */
extern int number;

int f(int x) {
    int y;
    y = x*SCALE + number;
    return (y);
}
```

Με αρχείο επικεφαλίδας

scope.c

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

int number;

int main() {
    int y;
    number = 5;
    y = f(4*SCALE);
    printf("%d,%d\n",number,y);
    return 0;
}
```

func.c

```
#include "myscope.h"

int f(int x) {
    int y;
    y = x*SCALE + number;
    return (y);
}
```

myscope.h

```
#define SCALE 2
int f(int param);
extern int number;
```

❖ Χρήση συναρτήσεων από το ένα στο άλλο

- Δήλωση prototype
- Εναλλακτικά, δημιουργία include file με prototypes
- Γενικός ρόλος που παίζουν τα include files?

❖ Πώς τα κάνω compile?

- Είτε όλα μαζί:
 - ✧ gcc *.c
- Είτε με χρήση **Makefile** γιατί πάντα ελέγχει τι έχει αλλάξει

Με Makefile

scope.c

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

int number;

int main() {
    int y;
    number = 5;
    y = f(4*SCALE);
    printf("%d,%d\n",number,y);
    return 0;
}
```

func.c

```
#include "myscope.h"

int f(int x) {
    int y;
    y = x*SCALE + number;
    return (y);
}
```

myscope.h

```
#define SCALE 2
int f(int param);
extern int number;
```

Makefile

```
scope: scope.o func.o myscope.h
        gcc -o scope scope.o func.o

func.o: func.c myscope.h
        gcc -c func.c

scope.o: scope.c myscope.h
        gcc -c scope.c
```

scope.c

Makefile

```
scope: scope.o func.o myscope.h  
        gcc -o scope scope.o func.o  
  
func.o: func.c myscope.h  
        gcc -c func.c  
  
scope.o: scope.c myscope.h  
        gcc -c scope.c
```

Κανόνας

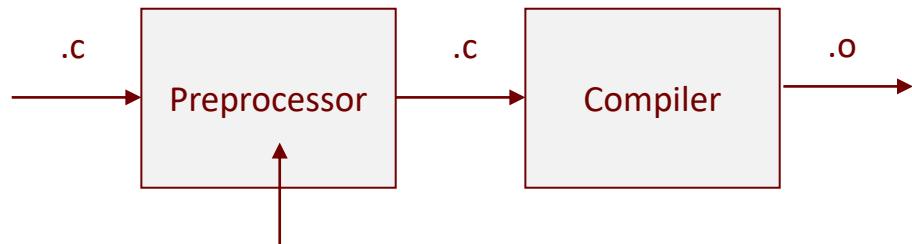
στόχος (target)

αρχεία απ' τα οποία εξαρτάται (προαπαιτούμενα)

εντολή / ενέργεια για να γίνει (η γραμμή αρχίζει με TAB!!!)

ελεογμ \ ελεύθερα λατ. λαγερ (μ λαστικημ αεβχρζερ λε TAB!!!)

Για κάθε αρχείο : **gcc -c x.c**



αντικαθιστά #include και #define

Αρχεία .o

άλλα αρχεία .o

gcc -o scope.exe scope.o f.o



Η γλώσσα C

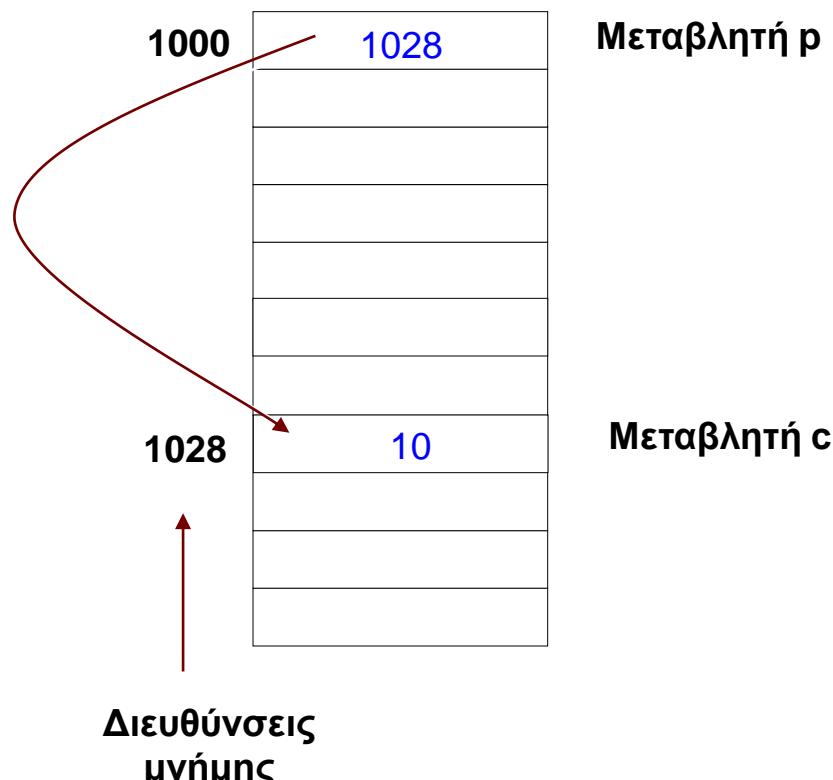
Δείκτες (*pointers*)



MYY502

Δείκτες - Pointers

- ❖ Δείκτης: τι είναι;
 - Μια μεταβλητή που περιέχει τη διεύθυνση μιας άλλης μεταβλητής



int c = 10;

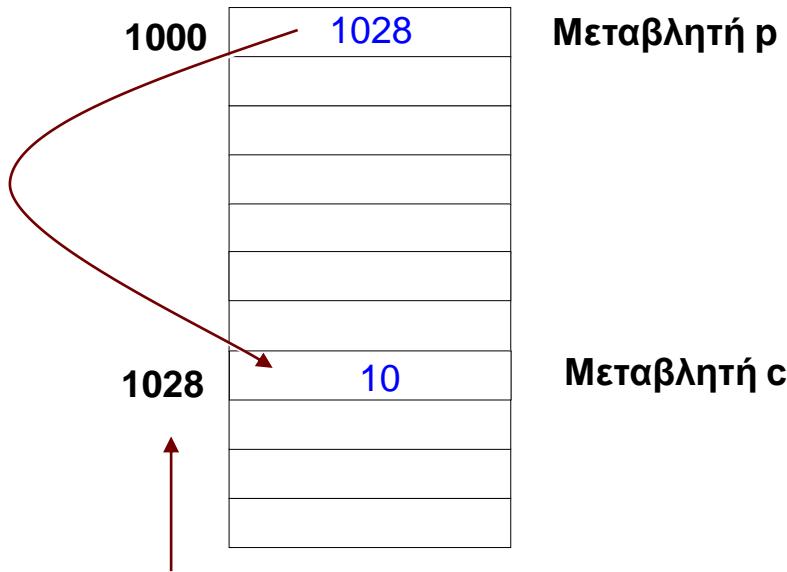
int *p;

p = &c;

*p : προσπέλαση του αντικειμένου που δείχνει ο δείκτης p.

Δείκτες - Pointers

- ❖ Τα περιεχόμενα του κελιού στο οποίο δείχνει ο δείκτης p δίνονται από το *p
- ❖ Τα περιεχόμενα του p = η διεύθυνση του κουτιού
- ❖ Η διεύθυνση στην οποία βρίσκεται ο c: &c



```
int c = 10;  
int *p;
```

```
p = &c;
```

*p : προσπέλαση του αντικειμένου που δείχνει ο δείκτης p.

Μεταβλητές & μνήμη

```
#include <stdio.h>

int main() {
    int var1;
    char var2[10];

    printf("Address of var1 variable: %x\n", &var1 );
    printf("Address of var2 variable: %x\n", &var2 );
    return 0;
}
```

Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6



Μεταβλητές & μνήμη

```
#include <stdio.h>

int main() {
    int var = 20; /* actual variable declaration */
    int * ip;      /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/
    printf("Address of var variable: %x\n", &var );
    printf("Address stored in ip variable: %x\n", ip );
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20

Αναλογία με αποθήκη προϊόντων

- ❖ Γνωστό κατάστημα διαθέτει αποθήκη με πολλά ράφια όπου σε κάθε ράφι υπάρχει ένα προϊόν (έπιπλο). Τα προϊόντα είναι συσκευασμένα έτσι ώστε ΔΕΝ ΜΠΟΡΕΙΣ να καταλάβεις τι είναι αν πας στα ράφια της αποθήκης. Για να πάρεις το έπιπλο πρέπει να επισκεφτείς την έκθεση του καταστήματος και να σημειώσεις σε ειδικά χαρτάκια το ΡΑΦΙ της αποθήκης που έχει το προϊόν που σε ενδιαφέρει. Με το συμπληρωμένο χαρτάκι πας στο σωστό ράφι και το παίρνεις – αλλιώς δεν μπορείς να βρεις το προϊόν.
- ❖ Αναλογία:
 - **Μνήμη** = αποθήκη
 - **Μεταβλητή** (κελί στη μνήμη) = το ράφι (ο αριθμός του)
 - **Τιμή μεταβλητής** = προϊόν στο ράφι
 - **Δείκτης** = το χαρτάκι



Έννοιες & αναλογίες

Λειτουργία / έννοια	Αναλογία
$p = \&x;$	Γράφω σε χαρτάκι το ράφι.
$*p$	Το προϊόν που υπάρχει στο ράφι («ταξιδεύω» εκεί και το βρίσκω).
$q = \&y;$	Άλλο χαρτάκι που γράφει άλλο ράφι.
$p = q;$	Στο πρώτο χαρτάκι αλλάζω τι έγραψα και γράφω το ίδιο ράφι που γράφει το δεύτερο χαρτάκι.
$temp = *a;$ $*a = *b;$ $*b = *a;$	«Ταξιδεύω» στα ράφια που γράφουν τα χαρτάκια a και b και εναλλάσσω τα προϊόντα.



- ❖ Δήλωση μεταβλητής: <type> *<name>;
- ❖ Παράδειγμα

```
int x, y;
```

```
int * p;
```

```
y = 3;
```

```
p = &y;
```

```
x = *p + 1;
```

- ❖ Προσοχή: η έκφραση **a*b** δεν περιλαμβάνει δείκτη!



❖ Σωστή αρχικοποίηση

```
int c;  
int * p;  
p = NULL; /* Χρειάζεται το stdio.h */  
p = 0; /* Ισοδύναμο με NULL */  
p = &c;
```

❖ Λάθος

```
p = 100;  
p = c;
```

Δήλωση μαζί με αρχικοποίηση δεικτών

- ❖ Μπορείτε να δηλώσετε έναν δείκτη και ταυτόχρονα να τον αρχικοποιήσετε:

```
int c, * p = &c; /* μην το κάνετε, μπερδεύει!! */
```

- Είναι ισοδύναμο με:

```
int c, * p;  
p = &c;
```

- ❖ ΠΡΟΣΟΧΗ:

- Στις δηλώσεις το «*p» σημαίνει ότι το p είναι δείκτης. ΔΕΝ σημαίνει ότι πηγαίνει εκεί που δείχνει και παίρνει το περιεχόμενο!
- Μόνο όταν το «*p» εμφανίζεται μέσα σε πράξεις έχει την έννοια του «πηγαίνω και παίρνω το περιεχόμενο»

- ❖ Για να μην υπάρχει σύγχυση, καλύτερα να μην αρχικοποιείτε με αυτόν τον τρόπο τους δείκτες. Το σωστότερο είναι να τους αρχικοποιείτε πάντα στην τιμή NULL και μετά να κάνετε ότι τροποποιήσεις θέλετε:

```
int c, * p = NULL;  
p = &c;
```

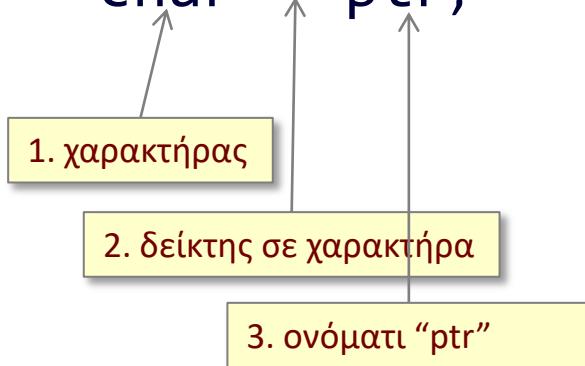
Δήλωση δείκτη

- ❖ Παράδειγμα:

```
char * ptr; /* Ισοδύναμα: char* ptr, char *ptr */
```

- ❖ «Διαβάζοντας» τη δήλωση:

```
char * ptr;
```



Πρόκειται για μία **μεταβλητή** που ονομάζεται `ptr`.

Η μεταβλητή είναι **δείκτης**.

Η θέση στην κύρια μνήμη στην οποία θα δείχνει, αποθηκεύει έναν **char**.

ελαλ υπακ

ουτογια θα ρετχλεγ' αυτοθμκερεγ

Δείκτες και πέρασμα παραμέτρων

❖ Παραδείγματα:

```
void foo(char *s);
```

```
void bar(int *result);
```

❖ Γιατί να περάσω δείκτη;

- Κλήση δια αναφοράς (call by reference)
- Αν θέλω, δηλαδή, η συνάρτηση να κάνει αλλαγές που να επηρεάζουν τα πραγματικά δεδομένα-ορίσματά της (δηλ. τις μεταβλητές της καλούσας συνάρτησης)
- Επίσης, αν θέλω να περάσω ως όρισμα μία μεγάλη δομή αποφεύγοντας τη χρονοβόρα αντιγραφή στη στοίβα (αργότερα αυτά)

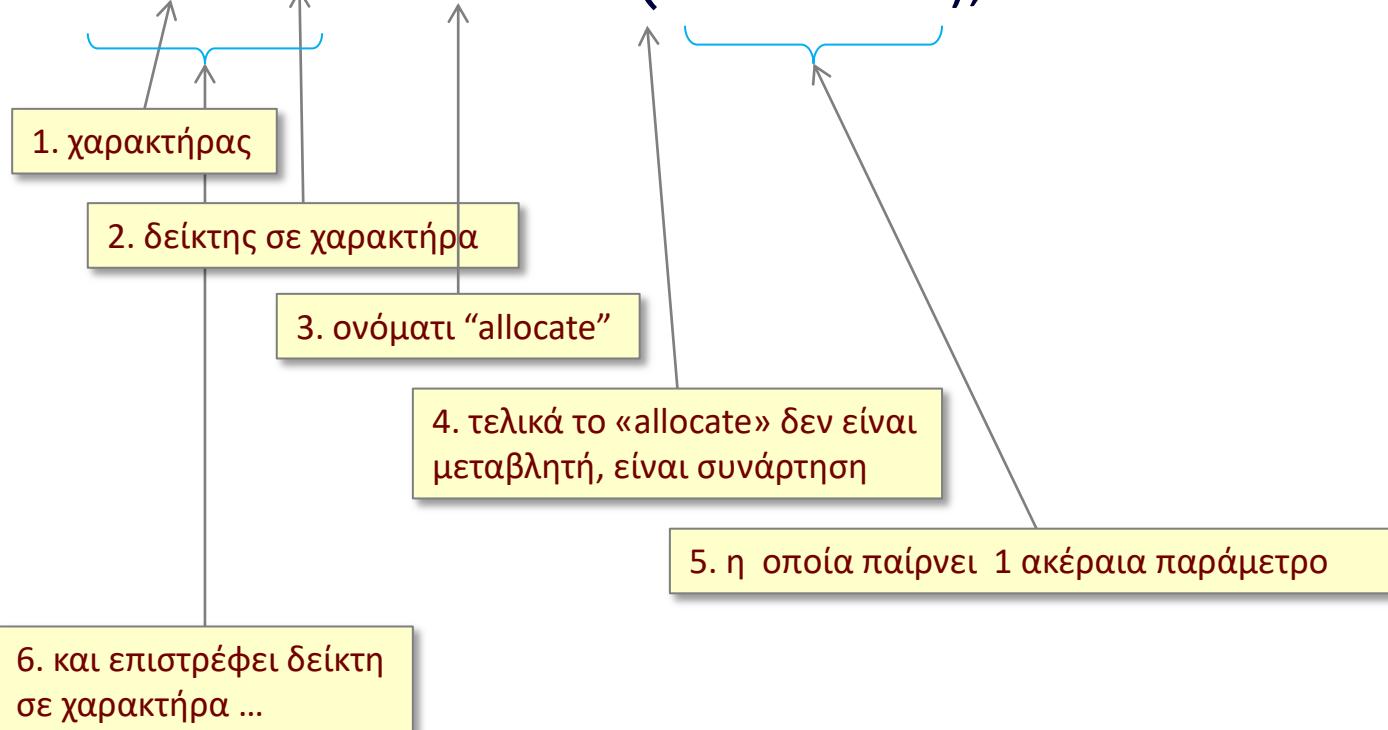
Συναρτήσεις που επιστρέφουν δείκτη

❖ Παράδειγμα:

```
char * allocate(int size);
```

❖ «Διαβάζοντας» τη δήλωση:

```
char * allocate(int size);
```



Ερωτήματα

❖ Υποθέστε τη δήλωση «`int x;`»

Τι ακριβώς κάνουν τα παρακάτω αν βρεθούν σε μια έκφραση;

- `&x;`
- `*(&x);`
- `&(*x);`

❖ Τι μεταβλητή χρειάζομαι για να βάλω μέσα τη διεύθυνση ενός δείκτη;

Κι άλλα ερωτήματα...

```
int i = 3, j = 5, k, * p=NULL, * q=NULL, * r=NULL;  
p = &i; q = &j;
```

* * & p

ισοδύναμο: $\ast(\ast(\&p))$

αποτέλεσμα: 3 (αφού είναι ισοδύναμο με $\ast(p)$)

3 * - * p / * q + 7

ισοδύναμο: $((3 \ast (- (*p))) / (*q)) + 7$

αποτέλεσμα: 6

3 * - * p /* q + 7

ισοδύναμο: λάθος! Το /* ξεκινά σχόλιο!!

* (r = & k) = *p * * q

ισοδύναμο: $(\ast(r = \&k)) = ((\ast p) * (\ast q))$

αποτέλεσμα: 15

Προτεραιότητες τελεστών C

Καλύτερα να μην θυμάστε τη σειρά!

Να καθορίζετε μόνοι σας τη σειρά των πράξεων βάζοντας στα σωστά σημεία

ΠΑΡΕΝΘΕΣΕΙΣ.

Operator	Description	Associativity
()	Function call	Left to right
[]	Array element	
->	Structure member pointer reference	
.	Class, structure or union member reference	
sizeof	Storage size in bytes of object / type	
++	Postfix Increment	
--	Postfix Decrement	
++	Prefix Increment	Right to left
--	Prefix Decrement	
-	Unary minus	
+	Unary plus	
!	Logical negation	
~	One's complement	
&	Address of	
*	Indirection	
(type)	Type conversion (cast)	Right to left
*	Multiplication	Left to right
/	Division	
%	Modulus	
+	Addition	Left to right
-	Subtraction	
<<	Bitwise left shift	Left to right
>>	Bitwise right shift	
<	Scalar less than	Left to right
<=	Scalar less than or equal to	
>	Scalar greater than	
>=	Scalar greater than or equal to	
==	Scalar equal to	Left to right
!=	Scalar not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
	Logical inclusive OR	Left to right
? :	Conditional expression	Right to left
=	Assignment	Right to left
+= -= *=	Assignment	
/= %= &=	Assignment	
^= =	Assignment	
<<= >>=	Assignment	
,	Comma	Left to right

Πέρασμα παραμέτρων με αναφορά – swap

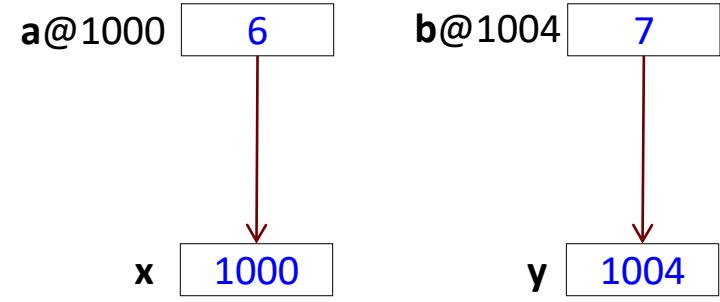
```
#include <stdio.h>
void swap (int x, int y);
int main() {
    int a, b;
    a = 6;
    b = 7;
    swap(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap (int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
    return;
}
```

Πέρασμα παραμέτρων με αναφορά – swap

```
#include <stdio.h>
void swap (int x, int y);
int main() {
    int a, b;
    a = 6;
    b = 7;
    swap(a, b);           → swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap (int x, int y) {           void swap (int *x1, int *x2) {
    int temp;
    temp = x;             }
    x = y;               →   { temp = *x1;
    y = temp;             *x1 = *x2;
    return;               *x2 = temp;
}
```

Swap με δείκτες

```
#include <stdio.h>
void swap (int *x, int *y);
int main() {
    int a, b;
    a = 6;
    b = 7;
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap (int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    return;
}
```



Τα **x**, **y** περιέχουν διευθύνσεις

Swap με δείκτες – εναλλακτικός κώδικας

```
#include <stdio.h>
void swap (int *x, int *y);
int main() {
    int a, b;
    a = 6;
    b = 7;
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
void swap (int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    return;
}
```

```
#include <stdio.h>
void swap (int *x, int *y);
int main() {
    int a = 6, b = 7, *pa, *pb;
    pa = &a;
    pb = &b;
    swap(pa, pb);
    printf("%d %d\n", a, b);
    return 0;
}
void swap (int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    return;
}
```

Παράδειγμα

```
#include <stdio.h>
int max(int x, int y);

int main() {
    int a, b, res;
    a = 5;
    b = 3;
    res = max(a, b);
    return 0;
}

int max(int x, int y) {
    int res;
    if (x > y) res = x;
    else res = y;
    return res;
}
```

```
#include <stdio.h>
void max(int x, int y, int * res);

int main() {
    int a, b, res;
    a = 5;
    b = 3;
    max(a, b, &res);
    return 0;
}

void max(int x, int y, int *res) {
    if (x > y) *res = x;
    else *res = y;
}
```

Παράδειγμα

```
#include <stdio.h>
void init(int * x, int * y);

int main() {
    int a, b;
    init(&a, &b);
    return 0;
}

void init(int *x, int *y) {
    *x = 12;
    *y = 15;
}
```

Πέρασμα παραμέτρων με αναφορά - πίνακες

- ❖ Αν θυμάστε, το πέρασμα των πινάκων σε μία συνάρτηση γίνεται πάντα με αναφορά.
 - Αυτό που γίνεται είναι ότι περνιέται **ΜΟΝΟ ΕΝΑΣ ΔΕΙΚΤΗΣ ΣΤΟ ΠΡΩΤΟ ΣΤΟΙΧΕΙΟ ΤΟΥ ΠΙΝΑΚΑ** (μπορείτε να φανταστείτε γιατί;;)
 - Τα παρακάτω είναι ισοδύναμα:

```
void initzero(int x[100]) {    // Αγνοείται το μέγεθος!
    x[0] = 10;
}
void initzero(int x[]) {        // Άρα δεν χρειάζεται
    x[0] = 10;
}
void initzero(int *x) {        // Είναι απλός δείκτης
    *x = 10;
}
```

- ❖ Μόνο σε παράμετρο συνάρτησης επιτρέπεται να μην δοθεί το πλήθος των γραμμών ενός πίνακα διότι ΑΓΝΟΕΙΤΑΙ – ο πίνακας περνιέται ως ένας απλός δείκτης, χωρίς πληροφορία για το πλήθος των στοιχείων του πίνακα.

Παράδειγμα

```
#include <stdio.h>
void initarr(int x[10]);

int main() {
    int a[10];
    initarr(a);
    return 0;
}

void initarr(int x[10]) {
    int i;
    for (i = 0; i < 10; i++)
        x[i] = i;
}
```

```
#include <stdio.h>
void initarr(int x[], int n);

int main() {
    int a[10];
    initarr(a, 10);
    return 0;
}

void initarr(int x[], int n) {
    int i;
    for (i = 0; i < n; i++)
        x[i] = i;
}
```

Παρατήρηση

- ❖ Δεν έχει νόημα μια συνάρτηση να επιστρέφει δείκτη που δείχνει τοπικά, δηλαδή τη διεύθυνση μιας τοπικής μεταβλητής:

```
int * test1() {  
    int i;  
    return &i;  
}
```

- ❖ Έχει διαφορά η παρακάτω περίπτωση;

```
int * test2() {  
    static int i;  
    return &i;  
}
```

Παρένθεση - υπενθύμιση

- ❖ Το α και β παρακάτω είναι ίδια; Αν όχι υπάρχει κάποιο πρόβλημα;

(a)	(b)
<code>int x, * y = &x;</code>	<code>int x, * y; *y = &x;</code>

- ❖ Θυμηθείτε ότι άλλο σημαίνουν οι τελεστές της C μέσα σε μία ΔΗΛΩΣΗ και άλλο μέσα σε μία εντολή/πράξη. Το (b) λοιπόν είναι διαφορετικό (και λάθος). Το (a) θα ήταν ισοδύναμο με το (c):

(a)	(c)
<code>int x, * y = &x;</code>	<code>int x, * y; y = &x;</code>

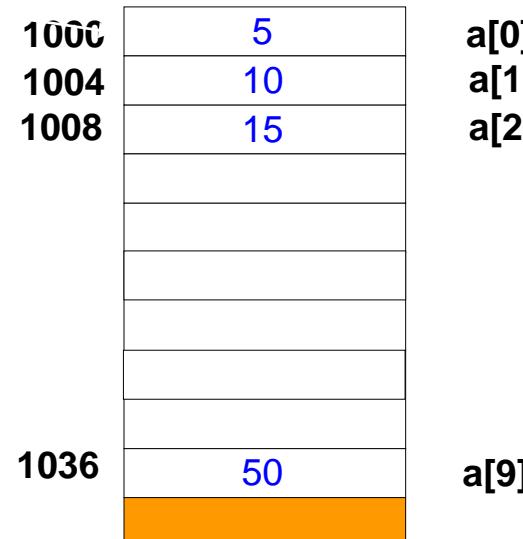
- ❖ Καλό είναι να μην κάνουμε αρχικοποίηση ενός δείκτη κατά τη δήλωσή του (παρά μόνο με *NULL*).

Δείκτες και Πίνακες

❖ Παράδειγμα:

```
int * p=NULL;  
int a[10] = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50};  
p = &a[0]; /* διεύθυνση 1ου στοιχείου του πίνακα */  
p = a; /* ισοδύναμο με το παραπάνω */  
p[i] <=> a[i] ⇔ *(p+i)
```

❖ Τι είναι η μεταβλητή a? Είναι ένας σταθερός δείκτης που αντιστοιχεί σε μια διεύθυνση. Ο σταθερός δείκτης δεν μπορεί να δείξει κάπου αλλού.



Ο τελεστής []

- ❖ Μέχρι τώρα γνωρίζαμε ότι ο τελεστής [] χρησιμοποιείται για επιλογή κάποιου στοιχείου ενός πίνακα, π.χ. το $a[3]$ υποδηλώνει το 3^o στοιχείο του πίνακα.
- ❖ Όπως αυτή είναι η «μισή» αλήθεια:
 - Βασικά το $a[3]$ σημαίνει το στοιχείο που είναι 3 θέσεις μετά από εκεί που δείχνει (ξεκινά) ο δείκτης a.
 - Επομένως, μπορεί να χρησιμοποιηθεί με γενικούς δείκτες και όχι μόνο με πίνακες!
 - Π.χ. αν $p = &a[0]$, μπορώ να πω $p[1] = 4$;
 - Π.χ. αν $p = &a[2]$, μπορώ να πω $p[1] = 4$;
- ❖ Η έκφραση **A[B]** είναι ισοδύναμη με ***(A+B)**
 - ✧ “Μυστικό”: επομένως το $a[2]$ είναι ισοδύναμο με το 2[a] !!!

Παράδειγμα

```
#include <stdio.h>
main() {
    int src[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int * p=NULL, * q=NULL;
    int n = 3;

    p = & src[0];
    q = & src[2];
    printf("%d %d\n", *p, p[0]);
    printf("%d %d\n", *q, q[0]);

    p[n] += 11;
    printf("%d\n", p[n]);

    q[n] += 11;
    printf("%d\n", q[n]);
}
```

Εκτυπώσεις:

1 1

3 3

15

17

❖ Μπορούμε να έχουμε εκφράσεις με δείκτες

- $p < q$ Ποιος δείχνει πιο μπροστά και ποιος πιο πίσω?
- $p + n$ Το κελί που βρίσκεται η κελιά πιο μετά
- $p - n$ Το κελί που βρίσκεται η κελιά πιο πριν
- $p - q$ Πόσο μακριά βρίσκονται (σε κελιά)?

❖ Άλλες δυνατές εκφράσεις

- $p = p + n$
- $p += n$

❖ Δεν δουλεύουν:

- $p + q$
- p^*4

Αριθμητική δεικτών

```
char c, * pc=NULL;  
int i, * pi=NULL;  
  
pc = &c; pi = &i;  
printf("pc = %x, pc+1 = %x", pc, pc+1);  
printf("pi = %x, pi+1 = %x", pi, pi+1);
```

Δίνει:

```
pc = 251af3c8, pc+1 = 251af3c9  
pi = 251af3c4, pi+1 = 251af3c8
```

Γιατί;

Διότι το +1 δεν είναι +1 byte αλλά +1 τύπος (όσα bytes πιάνει ο τύπος του δείκτη)

```
printf("char: %d bytes", sizeof(char));  
printf("int: %d bytes", sizeof(int));
```

Δίνει:

```
char: 1 bytes  
int: 4 bytes
```

Παράδειγμα

```
#include <stdio.h>
main() {
    int src[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int * p=NULL, * q=NULL;
    int n = 3;

    p = & src[0];
    q = & src[2];
    printf("%d %d\n", *p, p[0]);
    printf("%d %d\n", *q, q[0]);

    *(p+n) += 11;
    printf("%d\n", p[n]);
    .....  

    *(q+n) += 11;
    printf("%d\n", q[n]);
}

}
```

Εκτυπώσεις:

1 1	
3 3	
15	
17	

Συνεχίζεται...

Παράδειγμα

```
#include <stdio.h>
main() {
    int src[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int * p=NULL, * q=NULL;
    int n = 3;           ... από πριν
.....
if (q > p)
    printf("%d\n", q - p);           2
else
    printf("%d\n", p - q);

p++;
q+=2;

printf("%d %d\n", *p, p[0]);           2 2
printf("%d %d\n", *q, q[0]);           5 5
}
```

Εκτυπώσεις:

Δείκτες και πίνακες, πάλι

- ❖ Υποθέτουμε ότι έχουν δηλωθεί: `int a[10], *p;`
- ❖ **Ισχύει:**

$$\&a[i] \Leftrightarrow a+i$$

$$a[i] \Leftrightarrow *(a+i)$$

- ❖ Αν έχω εκτελέσει την εντολή: `p = a`, τότε **Ισχύει:**

$$\&a[i] \Leftrightarrow a+i \Leftrightarrow \&p[i] \Leftrightarrow p+i$$

$$a[i] \Leftrightarrow *(a+i) \Leftrightarrow p[i] \Leftrightarrow *(p+i)$$

- ❖ Ενώ ισχύει η έκφραση `p++`, **δεν ισχύουν** οι εκφράσεις: **a++** και **a = p**,
 - παρά μόνο αν το `a[]` είναι παράμετρος σε μια συνάρτηση

Παράδειγμα, πάλι

```
#include <stdio.h>
void initarr(int x[], int n);

int main() {
    int a[10];
    initarr(a, 10);
    return 0;
}
```

```
void initarr(int x[], int n) {
    int i;
    for (i = 0; i < n; i++)
        x[i] = i;
}
```

```
#include <stdio.h>
void initarr(int *x, int n);

int main() {
    int a[10];
    initarr(a, 10);
    return 0;
}

void initarr(int *x, int n) {
    int i;
    for (i = 0; i < n; i++)
        x[i] = i;
}
```

Παρένθεση: ο τελεστής sizeof

- ❖ Το sizeof() (το οποίο δεν είναι συνάρτηση αλλά **ΤΕΛΕΣΤΗΣ** της C), επιστρέφει το μέγεθος σε BYTES είτε ενός τύπου είτε μίας μεταβλητής:
- ❖ Τι θα τυπωθεί παρακάτω (υποθέστε 32-μπιτο σύστημα);

```
char x, s[5], * p = NULL;
int y, a[10], * q = NULL;
p = &x; q = a;
printf("char: %d, int: %d", sizeof(char), sizeof(int));
    char: 1, int: 4
printf("x: %d, y: %d", sizeof(x), sizeof(y));
    x: 1, y: 4
printf("s: %d, a: %d", sizeof(s), sizeof(a));
    s: 5, a: 40
printf("p: %d, q: %d", sizeof(p), sizeof(q));
    p: 4, q: 4
printf("*p: %d, *q: %d", sizeof(*p), sizeof(*q));
    p: 1, q: 4
```

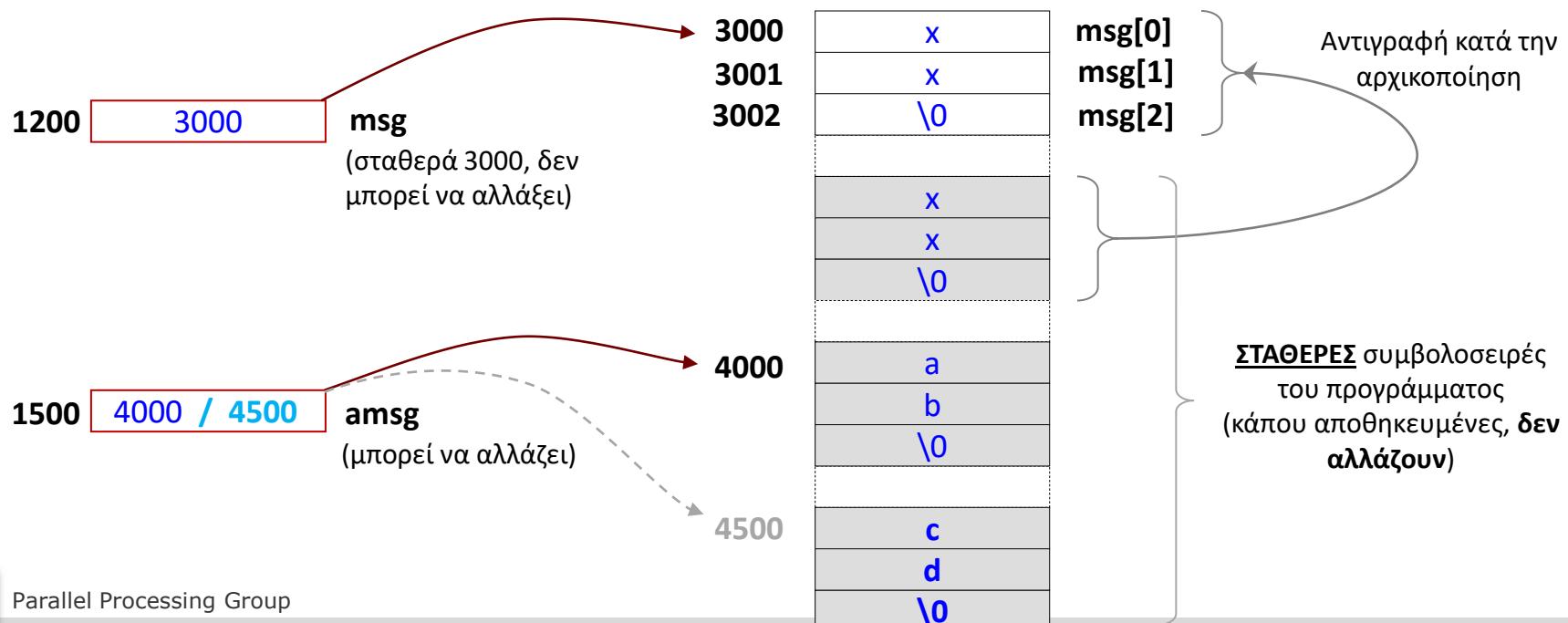
Τρικ με pointers

- ❖ Πώς μπορώ να δω αν το μηχάνημά μου αποθηκεύει με σειρά big endian ή little endian?
- ❖ *Homework ;-)*

Δείκτες και συμβολοσειρές

(ΥΠΕΝΘΥΜΙΣΗ: Κάθε string περιέχει και έναν παραπάνω χαρακτήρα, το '\0')

```
char msg[] = "xx"; /* Πίνακας 3 στοιχείων με αρχικοποίηση */
msg = "yyyy";        /* Δεν επιτρέπεται (σταθερός δείκτης)! */
char * amsg = "ab"; /* Δείκτης σε σταθερή συμβολοσειρά */
amsg = "cd";         /* OK - δείχνει σε άλλο χώρο! */
```



str.c

```
#include <stdio.h>
int main() {
    char buf[] = "hello";
    char * ptr = "geia sou";
    char * ptr1 = "file mou";
    buf[2] = '$';
    printf("%s\n", buf);

    ptr[2] = '%';           /* crash (why??) */

    ptr = ptr1;
    printf("%s\n", ptr);

    ptr[2] = '#';           /* crash (why??) */
    buf = ptr;              /* compile error (why??) */

    ptr = buf;
    ptr[2] = 'l';
    printf("%s\n", ptr);

    ptr = "bonjour";
    printf("%s\n", ptr);

    printf("buf occupies %d bytes in memory.\n", sizeof(buf)); /* ??? */
    return 0;
}
```

Παραδείγματα

```
#include <stdio.h>
void strcpy1(char s[], char d[]);

int main() {
    char x[30] = "abcdef";
    char y[10];
    strcpy1(x, y);
    printf("%s", y);
    return 0;
}

void strcpy1(char s[], char d[]) { /* Copy s to d */
    int i;
    i = 0;
    while (s[i] != '\0') {
        d[i] = s[i];
        i++;
    }
    d[i] = '\0';
}
```

Αντιγραφή – Εναλλακτικά

```
#include <stdio.h>
void strcpy2(char * s, char * d);

int main() {
    char x[30] = "abcdef";
    char y[10];
    strcpy2(x, y);
    printf("%s", y);
    return 0;
}

void strcpy2(char * s, char * d) { /* Copy s to d */
    while (*s != '\0') {
        *d = *s;
        s++; d++;
    }
    *d = '\0';
}
```

Αντιγραφή – Εναλλακτικά

```
void strcpy3 (char s[], char d[]) {  
    int i = 0;  
    while ((d[i] = s[i]) != '\0') {  
        i++;  
    }  
}
```

```
void strcpy4 (char * s, char * d) {  
    while ((*d = *s) != '\0') {  
        s++; d++;  
    }  
}
```

Σύγκριση

```
int strcmp1(char s1[], char s2[]) {  
    int i;  
    for (i = 0; s1[i] == s2[i]; i++) {  
        if (s1[i] == '\0') /* και τα δύο ίσα με \0 */  
            return 0;  
    }  
    return (s1[i] - s2[i]);  
}
```

- ❖ Η συνάρτηση επιστρέφει: 0 αν ΙΣΑ, < 0 αν $s_1 < s_2$, > 0 αν $s_1 > s_2$.
- ❖ Παράδειγμα: $s_1 \rightarrow "abc"$, $s_2 \rightarrow "abcd"$

Σύγκριση - Εναλλακτικά

```
int strcmp2(char * s1, char * s2) {
    for (; *s1 == *s2; s1++, s2++) {
        if (*s1 == '\0') {
            return 0;
        }
    }
    return *s1 - *s2;
}

int main() {
    char A[N], B[N];
    ...
    strcmp2(A, B);
    ...
}
```

```
int strlen1(char s[]) {  
    int i = 0;  
    while (s[i] != '\0') i++;  
    return i;  
}  
  
int strlen2(char * s) {  
    char *p = s;  
    while (*p != '\0') p++; /* προχωράει 1 byte */  
    return p-s ;  
}
```

Πίνακες Δεικτών (Array of Pointers)

- ❖ Τι είναι;
 - Πίνακας που περιέχει δείκτες
- ❖ Πώς δηλώνεται;

```
<type> * <name> [size];
```
- ❖ Πώς αρχικοποιείται;
 - `int A[10];`
 - `int B[20];`
 - `int * C[2] = { A, B };`
- ❖ Αντί για πίνακα 2 διαστάσεων χρησιμοποιώ πίνακα δεικτών
 - Οικονομία μνήμης όταν η 2^η διάσταση μεταβάλλεται
 - Όταν η 2^η διάσταση είναι άγνωστη

- ❖ Πίνακας με strings διαφορετικών μεγεθών

```
char * months[12] = {"January", "February", "March", ..};
```

Παράδειγμα

```
char * get_month(int i) {  
    static char * months[12] = {"Jan", "Feb", ...};  
    return ( (i<12 && i >=0) ? months[i] : NULL );  
}
```

- ❖ Τι είναι η τιμή NULL που μπορεί να επιστρέψει η συνάρτηση?
- ❖ Τι θα συμβεί αν βγάλω το static;

Επανάληψη – υπενθύμιση: τι δουλεύει και τι όχι;

```
#include <stdio.h>
int main() {
    char buf[] = "hello";                                // Αντιγραφή από σταθερή συμβολοσειρά
    char * ptr = "hello";                                // Δείχνει στον 1ο χαρακτήρα της σταθερ. συμβολ.
    void test1(char s[]);
    void test2(char *s);

    buf = ptr;                                         // Δεν επιτρέπεται να αλλάξει ο «δείκτης» buf
    ptr = buf;                                         // OK, ο δείκτης ptr θα δείχνει όπου και ο buf
    buf = "dolly";                                     // Δεν επιτρέπεται να αλλάξει ο δείκτης buf
    ptr = "dolly";                                     // OK, θα δείχνει όπου είναι η αρχή του "dolly"
    *buf = 'D';                                       // OK, ισοδύναμο με buf[0] = 'D';
    *ptr = 'D';                                       // Δεν επιτρέπεται αλλαγή στη σταθερή συμβολοσ.
    printf("%d", sizeof(buf));                         // 6 (5 χαρακτήρες + το \0)
    printf("%d", sizeof(ptr));                         // 4 (σε 32-μπιτο, 4 bytes για αποθήκευση δείκτη)
    test1(buf);
    test1(ptr);
    test2(buf);
    test2(ptr);
    return 0;
}

void test1(char s[]) {                                // Πίνακας-παράμετρος περνιέται ως απλός δείκτης
    printf("%d, %c", sizeof(s), *(s+1));           // 4 (σε 32-μπιτο, 4 bytes για αποθήκευση δείκτη)
}

void test2(char * s) {                               // 4 (σε 32-μπιτο, 4 bytes για αποθήκευση δείκτη)
    printf("%d, %c", sizeof(s), s[1]);
```

Είναι όλα OK?

```
#include <stdio.h>
void max(int x, int y, int * res);

int main() {
    int a, b, res;
    a = 5;
    b = 3;
    max(a, b, &res);
    return 0;
}
```

```
void max(int x, int y, int *res) {
    if (x > y) *res = x;
    else *res = y;
}
```

```
#include <stdio.h>
void max(int x, int y, int * res);

int main() {
    int a, b, *res;
    a = 5;
    b = 3;
    max(a, b, res);
    return 0;
}
```

```
void max(int x, int y, int *res) {
    if (x > y) *res = x;
    else *res = y;
}
```

Η γλώσσα C

Διαχείριση Συμβολοσειρών



MYY502

Συμβολοσειρές (strings)

❖ Θυμόμαστε: τι είναι οι συμβολοσειρές;

- Πίνακας χαρακτήρων (με τη σύμβαση ότι στο τέλος έχει '\0').
 - ✧ char str[10]; /* Άρα 9 χαρακτήρες + το \0 */
- Με εύκολη αρχικοποίηση:
 - ✧ char str[10] = "This is";
- Και με διαχείριση που είναι πιο εύκολη από τη διαχείριση πινάκων, όπως θα δούμε.
 - ✧ Όμως μην ξεχνάμε ότι είναι πίνακες (άρα ισχύουν ότι και για αυτούς).

❖ Τι είναι τα παρακάτω:

char b[10] = "this";

Το b και το "this" είναι strings.

char *c;

Δεν είναι string! Είναι ένας ptr σε χαρακτήρα.

char *d = "that";

Το "that" είναι string (αποθηκευμένο κάπου στη μνήμη) και ο d είναι pointer στον πρώτο χαρακτήρα του d. Λέμε καταχρηστικά ότι και το d είναι string.

Αρχικοποίηση

```
int main() {
    char q[] = "First";
    char r[16] = { 'S', 'e', 'c', 'o', 'n', 'd', '\0' };
    char s[16] = "Third";
    char t[16];
    char *u = "Fifth";
    char *w;

    t = "Fourth"; /* Δεν επιτρέπεται! Πρέπει t[0]='F', t[1]='o' ... */
    w = "Sixth"; /* Επιτρέπεται, μιας και είναι pointer */
    return 0;
}
```

puts/fputs

❖ int puts(char *s);

- Η puts τυπώνει το string s και ένα χαρακτήρα νέας γραμμής ('\n') στην οθόνη.
- Επιστρέφει EOF αν συμβεί κάποιο λάθος, διαφορετικά επιστρέφει μη αρνητική τιμή.

❖ int fputs(char *s, FILE *fp);

- Η fputs τυπώνει το string s στο αρχείο fp (χωρίς επιπλέον \n).
- Για τύπωμα στην οθόνη, περάστε ως αρχείο το **stdout**.
- Επιστρέφει EOF αν συμβεί κάποιο λάθος, διαφορετικά επιστρέφει μη αρνητική τιμή.
- Π.χ. fputs("hi", stdout);



gets/fgets

- ❖ `char *gets(char *s);`
 - Η gets διαβάζει την επόμενη γραμμή εισόδου και την αποθηκεύει στο string s.
 - **Αντικαθιστά τον τερματικό χαρακτήρα νέας γραμμής με '\0'.**
 - Επιστρέφει s, ή NULL αν συναντηθεί το τέλος του αρχείου (EOF) ή αν συμβεί κάποιο λάθος.
 - **Δεν είναι ασφαλής συνάρτηση** (τι γίνεται αν το s[] δεν φτάνει?).

- ❖ `char *fgets(char *s, int num, FILE *fp);`
 - Η gets διαβάζει **μέχρι num-1 χαρακτήρες** από το αρχείο fp (το πληκτρολόγιο είναι το **stdin**) και την αποθηκεύει στο string s.
 - **Αν πληκτρολογήθηκε το newline, τότε γράφεται ΚΑΙ ο χαρακτήρας \n μέσα στο s και αμέσως μετά γράφεται και το '\0'.**
 - Επιστρέφει s, ή NULL αν συναντηθεί το τέλος του αρχείου (EOF) ή αν συμβεί κάποιο λάθος.
 - **Να την προτιμάτε έναντι της gets().**
 - Π.χ. `fgets(str,9,stdin); /* έως 8 χαρακτήρες από πληκτρολόγιο */`



printf / scanf

❖ Εκτύπωση ή διάβασμα με το "%s" στο format

❖ printf:

- "%s" : τυπώνει όλο το string
- "%Ns" τυπώνει τουλάχιστον N χαρακτήρες (με κενά αν χρειαστεί)
- "%.Ms" τυπώνει το πολύ M χαρακτήρες
- "%N.Ms" τυπώνει τουλάχιστον N και το πολύ M χαρακτήρες
- "%-Ns" τυπώνει τουλάχιστον N χαρακτήρες, με αριστερή στοίχιση
- Π.χ. printf("%-3.5s", str);

❖ scanf:

- "%s": διαβάζει μία συμβολοσειρά
- **ΠΡΟΣΟΧΗ:** η συμβολοσειρά τελειώνει όταν συναντηθεί χαρακτήρας κενού (*space, tab, newline*)



Παραδείγματα

```
/* gets example */
#include <stdio.h>
int main() {
    char string [256];
    printf ("Insert your full address: ");
    gets(string);
    printf("Your address is: %s\n", string);
    return 0;
}
```

```
/* puts example */
#include <stdio.h>
int main () {
    char string [] = "Hello world!";
    puts(string);
}
```

Παραδείγματα

```
#include <stdio.h>
int main() {
    char test1[5], test2[5];
    scanf("%s", test1);
    printf("test1=%s\n", test1);
    gets(test2);
    printf("test2=%s\n", test2);
    return 0;
}
```

Εκτέλεση:

\$./a.out

1313

test1=1313

test2=

\$...

To `test2` θα είναι ίσο με "end of line" από την προηγούμενη είσοδο διότι η `scanf()` ολοκληρώνει το διάβασμά της μόλις συναντήσει τον πρώτο κενό χαρακτήρα (space, tab, newline) – αλλά δεν τον «καταναλώνει»! Έτσι η `gets()` βρίσκει το newline και επιστρέφει αμέσως...



Παραδείγματα

```
#include <stdio.h>
int main() {
    char test1[5], test2[5];
    scanf("%s", test1);
    printf("test1=%s\n", test1);
    scanf("%s", test2);
    printf("test2=%s\n", test2);
    return 0;
}
```

Εκτέλεση:

\$./a.out

1313

test1=1313

1233

test2=1233

\$...

Χρησιμοποιείται η `scanf()` αντί για την `gets()`. Η `scanf(%s)` στην αρχή του διαβάσματος αγνοεί τα κενά (space, tab, newline) και άρα την αλλαγή γραμμής. Στη συνέχεια διαβάζει από το πρώτο μη-blank και σταματά να διαβάζει στο αμέσως επόμενο blank.

Παραδείγματα

```
#include <stdio.h>
int main() {
    char lula[]="lula";
    char *ptr = lula;
    puts(lula); puts(" > ");
    puts(ptr + 2);
    printf("%s > %s\n", lula, ptr+2);
    return 0;
}
```

Εκτέλεση:

```
$ ./a.out
Lula
>
La
Lula > La
$ ...
```

Η puts προσθέτει το \n

Παραδείγματα

- ❖ «Χειροποίητες» `puts` που δεν εκτυπώνουν το EOL.
 - Η `putchar(ch)` τυπώνει στην οθόνη τον χαρακτήρα `ch`
 - Είναι ισοδύναμη με το `printf("%c", ch);`

```
void myputs(char *string) {  
    int i = 0;  
    while (string[i] != '\0') putchar(string[i++]);  
}  
  
void myputs(char *string) {  
    while (*string != '\0') {  
        putchar(*string);  
        string++;  
    }  
}  
  
/* Το '\0' έχει ASCII κωδικό μηδέν (0) ! */  
void myputs(char *string) {  
    while (*string) putchar(*(string++));  
}
```

- ❖ `int sprintf(char *s, char *format, ...)`
 - Η συνάρτηση αυτή λειτουργεί ακριβώς όπως η printf() με τη διαφορά ότι **δεν τυπώνει στην οθόνη αλλά γράφει στο string s στο οποίο τοποθετείται επιπλέον και ο χαρακτήρας '\0'.**
 - Επιστρέφεται ο αριθμός των χαρακτήρων που γράφτηκαν στο s πλην του χαρακτήρα '\0'.
- ❖ `int sscanf(char *s, char *format, ...)`
 - Η συνάρτηση αυτή λειτουργεί ακριβώς όπως η scanf() με τη διαφορά ότι **η είσοδος των δεδομένων προέρχεται από το string s και όχι από το πληκτρολόγιο.**
 - Επιστρέφει είτε τον αριθμό των αντικειμένων που ενημερώθηκαν (αν όλα πάνε καλά), είτε EOF (σε περίπτωση λάθους).

Παράδειγμα

```
/* sprintf example */
#include <stdio.h>
int main () {
    char buffer[50];
    int n, a=5, b=3;

    n=sprintf(buffer, "%d plus %d is %d", a, b, a+b);
    printf("[%s] is a %d char string\n", buffer, n);
    return 0;
}
```

```
$ ./a.out
[5 plus 3 is 8] is a 13 char string
```

Παράδειγμα

```
#include <stdio.h>

int main() {
    int k, m;
    float f;
    char *x="2 minutes to 12.0";
    char y[20], z[20], w[80];

    sscanf(x, "%d%s%s%f", &m, y, z, &f) ;
    printf("%d\n%s\n%s\n%f\n", m, y, z, f) ;
    k = sprintf(w, "%d %s %d %s ",m, z, (int) f, y) ;
    printf("\nNew order: %s\n", w);
    printf("with %d characters (including spaces)\n", k);
    return 0;
}
```

\$./a.out

2

minutes

to

12.000000

New order: 2 to 12 minutes

with 15 characters (including spaces)

Επαναληπτική κλήση της sscanf

```
#include <stdio.h>

int main() {
    int k;
    char x[30] = "2 minutes to 12.0";
    char *p;
    char y[30];

    p = x;
    while (sscanf(p, "%s", y) > 0) {
        k = printf("%s\n", y); /* # chars in y, +1 */
        p = p + k;             /* skip k chars */
    }
    return 0;
}
```

Επαναληπτική μέτρηση λέξεων

```
#include <stdio.h>

int main() {
    int k, count;
    char x[30], y[30], *p;

    while (1) {
        count = 0;
        if (fgets(x, 30, stdin) == NULL) /* Ctrl-D */
            break;
        p = x;
        while (sscanf(p, "%s", y) > 0) {
            k = printf("%s\n", y);
            p = p + k;
            count++;
        }
        printf("Total number of words: %d\n", count);
    }
    return 0;
}
```

Επαναληπτική μέτρηση λέξεων (συντομότερη)

```
#include <stdio.h>

int main() {
    int count;
    char x[90], y[90], *p;

    while (fgets(x, 90, stdin) != NULL) {
        for (count = 0, p = x; sscanf(p,"%s", y) > 0; count++) {
            p = p + printf("%s\n", y);
        }
        printf("Total number of words: %d\n", count);
    }
    return 0;
}
```

Διαχείριση Συμβολοσειρών μέσω **string.h**

❖ Μια συμβολοσειρά (string) είναι ένας πίνακας χαρακτήρων στον οποίο τοποθετείται τελευταίος ο χαρακτήρας '\0', ως ένδειξη του τέλους της συμβολοσειράς

❖ Μπορούμε να διαχειριστούμε ένα string με δύο τρόπους

➤ Ως έναν πίνακα, το οποίο συνεπάγεται σχετική δυσκολία

```
char line[8];
line[0] = 'H'; line[1] = 'e'; line[2] = 'l'; line[3] =
'l'; line[4] = 'o'; line[5] = '\0';
```

➤ Μέσω της χρήσης ειδικών συναρτήσεων που παρέχει η C μέσω του αρχείου **<string.h>**

```
strcpy(line, "Hello");
```

Συναρτήσεις Διαχείρισης Συμβολοσειρών

❖ `char *strcpy(s, t);`

- Αντιγράφει το string `t` στο `s`, μαζί με τον χαρακτήρα '\0' και επιστρέφει το `s`

- Παράδειγμα

```
char s[6];
strcpy(s, "hello");
```

- Τι γίνεται αν στο `s` δεν χωράει το `t`;

❖ `char *strncpy(s, t, n);`

- Αντιγράφει το πολύ ο χαρακτήρες από το `t` στο `s`, το `t` μπορεί να έχει λιγότερους. Επιστρέφει το `s`

❖ `char *strcat(s, t);`

- Προσθέτει στο τέλος του `s` το string `t`. Επιστρέφει το `s`.

❖ `char *strncat(s, t, n);`

- Προσθέτει στο τέλος του `s` το πολύ ο χαρακτήρες του `t`, και τοποθετεί επίσης και τον χαρακτήρα '\0'. Επιστρέφει το `s`.

Συναρτήσεις Διαχείρισης Συμβολοσειρών

- ❖ `int strcmp(s, t);`
 - Συγκρίνει λεξικογραφικά τα δύο strings.
 - ✧ (βασικό κριτήριο) περιεχόμενο
 - ✧ (δευτερεύον κριτήριο) μήκος
 - Επιστρέφει:
 - ✧ Αν ($s == t$) → 0
 - ✧ Αν ($s > t$) → θετικό
 - ✧ Αν ($s < t$) → αρνητικό
- ❖ `int strncmp(s, t, n);`
 - Όπως και παραπάνω αλλά συγκρίνει λεξικογραφικά το πολύ η χαρακτήρες
- ❖ `char *strstr(s, t);`
 - Επιστρέφει ένα δείκτη στην πρώτη εμφάνιση στο s του t , (διαφορετικά) επιστρέφει NULL αν το t δεν περιέχεται στο s .
- ❖ `int strlen(s);`
 - Επιστρέφει το μήκος της συμβολοσειράς s (χωρίς το \0).

❖ Έστω:

- `char a[30] = "Kalimera, ";`
- `char b[20] = "Kalo mathima!" ;`

❖ Τότε:

- `strcpy(a,b)`
 - ✧ `printf("%s", a);` → *Kalo mathima!*
- `strncpy(a,b,4)`
 - ✧ `printf("%s", a);` → *Kalomera,*
- `strcat(a,b)`
 - ✧ `printf("%s", a);` → *Kalimera, Kalo mathima!*

❖ Έστω:

- int ret; char *p;
- char a[30] = "Kalimera, ";
- char b[20] = "Kalo mathima!";

❖ Τότε:

- ret = strcmp(a,b);
 - ✧ printf("%d", ret); → κάποια αρνητική τιμή
- p = strstr(a,"im");
 - ✧ printf("%s", p); → imera,
- printf("%d", strlen(a)); → 10

❖ Τι θα τυπώσει το: printf("%d, %d", sizeof(a), strlen(a));

- 30, 10

Συναρτήσεις Διαχείρισης Συμβολοσειρών

- ❖ `char *strtok(char *s, char *t);`
 - Ψάχνει στο s για κομμάτια (tokens) που διαχωρίζονται με τους χαρακτήρες που περιγράφονται στο t.
 - Κάθε διαφορετική κλήση της strtok επιστρέφει και ένα καινούργιο token (κανονικό string με χαρακτήρα τερματισμού).
 - Χρήση:
 - ✧ 1^o token: καλώ tok = strtok(s, t);
 - ✧ Επόμενα (συνήθως σε loop): καλώ tok = strtok(NULL, t);
 - Επιστρέφει NULL αν δεν υπάρχουν άλλα tokens στο s.

Προσοχή!

- **Η συνάρτηση τροποποιεί το πρώτο όρισμα της**
- **Δεν μπορεί να χρησιμοποιηθεί σε σταθερά strings**

Παράδειγμα strtok

```
char email[] = "zas@cse.uoi.gr";
char token[] = "@";
char *s;

s = strtok(email, token);
s = strtok(NULL, token);

...
```

Παράδειγμα strtok (1/2)

- ❖ Να γράψετε πρόγραμμα το οποίο λαμβάνει διευθύνσεις ηλεκτρονικού ταχυδρομείου και επιστρέφει τα πεδία από τα οποία αποτελούνται
- ❖ Σχετικό παράδειγμα εκτέλεσης:

```
$ myprog
```

```
type email address: zas@cse.uoi.gr
```

```
fields of email address: zas, cse, uoi, gr
```

Παράδειγμα strtok (2/2)

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

int main() {
    char email[80];
    char token[] = "@.";
    char *s;

    printf("type email address:");
    scanf("%s", email);
    printf("fields of email address:");
    s = strtok(email, token);
    if (s != NULL) {
        printf("%s", s);
    }
    while ((s = strtok(NULL, token)) != NULL) {
        printf(", %s", s);
    }
    return 0;
}
```

Συναρτήσεις Ελέγχου - Μετατροπής

- ❖ #include <ctype.h>
- ❖ Συναρτήσεις ελέγχου
 - int isalnum(int c); true για γράμμα ή ψηφίο
 - int isalpha(int c); true για γράμμα
 - int isdigit(int c); true για ψηφίο
 - int isspace(int c); true για κενό, tab, \n, ...
 - int islower(int c); true για γράμμα μικρό
 - int isupper(int c); true για γράμμα κεφαλαίο
- ❖ Συναρτήσεις μετατροπής
 - int tolower(int c); μετατροπή κεφαλαίου σε μικρό
 - int toupper(int c); μετατροπή μικρού σε κεφαλαίο

Συναρτήσεις Ελέγχου - Μετατροπής

❖ #include <stdlib.h>

- `int atoi(char *s);` μετατροπή string σε ακέραιο
 - ✧ Το string s πρέπει να ξεκινά με κενό ή κάποιον αριθμό
 - ✧ Η συνάρτηση σταματά να διαβάζει από το string μόλις βρει κάποιον μη-αριθμητικό χαρακτήρα
 - ✧ Αν η μετατροπή δεν μπορεί να συμβεί, επιστρέφει 0
- `long atol(char *s);` μετατροπή string σε long
- `double atof(char *s);` μετατροπή string σε double

❖ Πώς μετατρέπω αριθμούς σε strings;

- *Homework!*

Παραδείγματα

- ❖ `int i;`
`i = atoi("512");`
`i = atoi("512.035");`
`i = atoi(" 512.035");`
`i = atoi(" 512+34");`
`i = atoi(" 512 bottles of beer on the wall");`
- ❖ `int i = atoi(" does not work: 512"); // → i = 0`
- ❖ `long l = atol("1024.0001");`
- ❖ `double x = atof("42.0is_the_answer");`

- ❖ Βασικές συναρτήσεις εισόδου – εξόδου
 - `int printf(char *format, ...);`
 - `int scanf(char *format, ...);`
- ❖ Ειδικοί χαρακτήρες στο `format` τους
 - Ακέραιοι αριθμοί
 - ✧ `%d` στο δεκαδικό σύστημα
 - ✧ `%u` χωρίς πρόσημο στο δεκαδικό σύστημα
 - ✧ `%o` χωρίς πρόσημο στο οκταδικό σύστημα
 - ✧ `%x` χωρίς πρόσημο στο δεκαεξαδικό σύστημα
 - Αριθμοί κινητής υποδιαστολής
 - ✧ `%f` σε μορφή: [-]ddd.dddddd
 - ✧ `%e` σε μορφή: [-]ddd.dddddd e[+/-]ddd
 - ✧ `%g` σε μορφή %f ή %e

❖ Ειδικοί χαρακτήρες στο format τους

➤ Άλλοι τύποι

- ❖ %c χαρακτήρας
- ❖ %s συμβολοσειρά (string)
- ❖ %p δείκτης

❖ Παραλλαγές στο format

➤ Μέγεθος αριθμών

- ❖ %h αριθμοί short, π.χ. %hd, %hx
- ❖ %l αριθμοί long ή double, π.χ. %ld, %lf
- ❖ %L αριθμοί long double, π.χ. %Lf

❖ Παραλλαγές στο format

➤ Μήκος αποτελέσματος

- ✧ %8d αριθμός σε μήκος 8 χαρακτήρων
- ✧ %20s συμβολοσειρά σε μήκος 20 χαρακτήρων
- ✧ %+8d αριθμός σε μήκος 8 χαρακτήρων, τύπωσε και πρόσημο
- ✧ %08d αριθμός σε μήκος 8 χαρακτήρων, τα πρώτα εξ' αυτών 0
- ✧ %-8d όπως το %8d με στοίχιση αριστερά

printf demo (1/2)

```
#include <stdio.h>
int main() {
    int i;          // Number to print.
    double f;        // Number to print.

    printf("Enter an integer (use either + or -): ");
    scanf ("%d", &i);
    printf("This is the integer.....| %d|\n", i);
    printf("This is the integer in octal.....| %o|\n", i);
    printf("Octal with leading zero.....| %#o|\n", i);
    printf("This is the integer in hex.....| %x| or |%X|\n", i, i);
    printf("Hex with leading 0x.....| %#x| or |%#X|\n", i, i);
    printf("Forcing a plus or minus sign.....| +d|\n", i);
    printf("Include space before + numbers.....| % d|\n", i);
    printf("Field width of 3.....| %3d|\n", i);
    printf("Field width of 5.....| %5d|\n", i);
    printf("Field width of 7.....| %7d|\n", i);
    printf("Same as above with left justification.| %-7d|\n", i);
    printf("Field width of 7 with zero fill.....| %07d|\n", i);
    printf("At least 3 digits.....| %.3d|\n", i);
    printf("At least 5 digits.....| %.5d|\n", i);
    printf("Field width of 10, at least 7 digits..| %10.7d|\n", i);
```

printf demo (2/2)

```
printf("\nEnter a floating point number: ");
scanf ("%lf", &f);

printf("This is the number.....| %f|\n", f);
printf("Forcing a plus or minus sign.....| %+f|\n", f);
printf("Field width of 20.....| %20f|\n", f);
printf("0 decimal places.....| %.0f|\n", f);
printf("0 decimal places forcing decimal.....| %#.0f|\n", f);
printf("3 decimal places.....| %.3f|\n", f);
printf("20 decimal places.....| %.20f|\n", f);
printf("Field width of 20, 3 decimal places...| %20.3f|\n", f);
printf("\nHere is the number in e format:\n");
printf("3 decimal places.....| %.3e|\n", f);
printf("5 decimal places & big 'e'.....| %.5E|\n", f);

printf("\nHere is the number in g format:\n");
printf("No special requests.....| %g|\n", f);
printf("Maximum of 1 significant figure.....| %.1g|\n", f);
printf("Maximum of 4 significant figures.....| %.4g|\n", f);

return 0;
}
```

Συμπεριφορά `scanf()`

Τα παρακάτω περιγράφουν πλήρως την μερικές φορές «περίεργη» συμπεριφορά της `scanf()`:

- ❖ `scanf(%c ...)`

- Διαβάζει αμέσως όποιον χαρακτήρα βρει (ακόμα και κενό) και επιστρέφει

- ❖ `scanf(%οτιδήποτε άλλο ...)`

- Αγνοεί τα κενά (space, tab, newline) και αρχίζει και διαβάζει μόλις συναντήσει μη-κενό χαρακτήρα
 - Τελειώνει και επιστρέφει μόλις συναντήσει κενό χαρακτήρα. Όμως, δεν τον καταναλώνει.

- ❖ Διαχείριση χαρακτήρων/συμβολοσειρών
 - `int putchar(int c);`
 - `int getchar();`
 - `int puts(char *s); /* also: fputs() */`
 - `char *gets(char *s); /* unsafe, prefer: fgets() */`
- ❖ Διαχείριση συμβολοσειρών `<string.h>`
 - `size_t strlen(char *s);`
Μέτρηση αριθμού χαρακτήρων της συμβολοσειράς s
 - `char *strcpy(char *s1, const char *s2);`
Αντιγραφή της συμβολοσειράς s2 στην s1
 - `char *strcat(char *s1, const char *s2);`
Προσθήκη της συμβολοσειράς s2 στο τέλος της s1
 - `int strcmp(char *s1, const char *s2);`
Σύγκριση των συμβολοσειρών s1 και s2

❖ Μετατροπή συμβολοσειρών <stdlib.h>

- `int atoi(char *s):`
Μετατροπή της συμβολοσειράς `s` σε `int`.
- `long int atol(char *s):`
Μετατροπή της συμβολοσειράς `s` σε `long int`.
- `double atof(char *s):`
Μετατροπή της συμβολοσειράς `s` σε `double`.

Προγραμματισμός σε C

Περίπτωση διαχείρισης συμβολοσειρών:
Ορίσματα στη main()



Ορίσματα στην main()

```
$ ls  
$ ls -l; ls -al; gcc myfile.c -lm
```

- ❖ Γενικά σε ένα πρόγραμμα μπορούμε να δώσουμε ως είσοδο δεδομένα/ορίσματα/επιλογές τη στιγμή που ξεκινάει η εκτέλεση του, από τη γραμμή εντολών
- ❖ Πώς μπορούμε να γνωρίζουμε τα δεδομένα/ορίσματα που δίνει ο χρήστης;
 - Απάντηση:
Παράμετροι στην main()! (την οποία μέχρι τώρα την ορίζαμε χωρίς παραμέτρους)
- ❖ Τα δεδομένα τα δέχεται η main() ως strings

Ορίσματα στην main()

- ❖ Μέχρι τώρα:

```
int main() { ... }
```

- ❖ Γενικά όμως, ο προγραμματιστής μπορεί να γράψει:

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

ή ισοδύναμα:

```
int main(int argc, char **argv) { ... }
```

- ❖ Το argv είναι ένας πίνακας δεικτών σε συμβολοσειρές (strings)
- ❖ Το argc είναι το πλήθος των στοιχείων του πίνακα
- ❖ Πάντα το στοιχείο 0 είναι το όνομα του προγράμματος

Παράδειγμα ορισμάτων στην main()

\$ a.out hi there

- ❖ argc = 3
- ❖ argv[0]: όνομα προγράμματος, "a.out"
- ❖ argv[1]: πρώτο όρισμα προγράμματος "hi"
- ❖ argv[2]: δεύτερο όρισμα προγράμματος "there"

Παράδειγμα (εκτύπωση ορισμάτων)

```
$ a.out hello world  
argc = 3  
Program name: a.out  
Arguments: hello, world
```

```
#include <stdio.h>  
int main(int argc, char *argv[]) {  
    int i;  
    printf("argc = %d\n", argc);  
    printf("Program name: %s\n", argv[0]);  
    printf("Arguments: ");  
    for (i = 1; i < argc; i++)  
        printf("%s, ", argv[i]);  
    printf("\n");  
    return 0;  
}
```

Επιπλέον παράδειγμα (πρόσθεση ορισμάτων)

```
$ myadd 5 10 15
```

Program name: myadd

Result: 30

```
#include <stdio.h>
#include <stdlib.h>      /* because of atoi() */
int main(int argc, char *argv[]) {
    int i, sum = 0;
    printf("Program name: %s\n", argv[0]);
    if (argc < 2) exit(1);    /* Nothing to add */
    for (i = 1; i < argc; i++) {
        sum = sum + atoi(argv[i]);
    }
    printf("Result: %d\n", sum);
    return 0;
}
```

Η γλώσσα C

Δείκτες και Διαχείριση Μνήμης
(memory management)



MYY502

1 πείραμα = πολλές μετρήσεις

- ❖ Κατασκευάστε ένα πρόγραμμα το οποίο δέχεται ως είσοδο από το χρήστη τον συνολικό αριθμό μετρήσεων που έγιναν κατά τη διάρκεια ενός πειράματος
- ❖ Εν συνεχεία ο χρήστης εισάγει στο πρόγραμμα τις επιμέρους τιμές των μετρήσεων σε ένα πίνακα
- ❖ Τέλος το πρόγραμμα επεξεργάζεται τα δεδομένα
 - μεταξύ άλλων υπολογίζει τον μέσο όρο των μετρήσεων



Άγνωστο μέγεθος πίνακα

```
#include <stdio.h>
int main() {
    int n;
    int i;
    float mo = 0;

    scanf("%d", &n);
    float measurements[n];
    πώς σας φαίνεται αυτή η υλοποίηση?

    for(i=0; i<n; i++)
        scanf("%f", &measurements[i]);

    for(i=0; i<n; i++)
        mo += measurements[i];

    mo = mo / n;
    ...
}
```

Άγνωστο μέγεθος πίνακα

```
#include <stdio.h>
int main() {
    int n;
    int i;
    float mo = 0;

    scanf("%d", &n);
    float measurements[n];

    for(i=0; i<n; i++)
        scanf("%f", &measurements[i]);

    for(i=0; i<n; i++)
        mo += measurements[i];

    mo = mo / n;
    ...
}
```

η υλοποίηση είναι λάθος!!

α) οι δηλώσεις πρέπει να γίνονται **ΜΟΝΟ** στην αρχή του μπλοκ { ... }

β) ΔΕΝ επιτρέπεται να είναι μεταβλητό το μέγεθος του πίνακα!

Σημείωση:

Η C99 τα επιτρέπει και τα δύο (δηλώσεις οπουδήποτε και “variable length arrays”).

Δυναμική δέσμευση μνήμης (dynamic memory allocation)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n; int i; float mo = 0;
    float *measurements;
    scanf("%d", &n);

    /* Μνήμη για n μετρήσεις */
    measurements = (float *) malloc(n * sizeof(float));
    if (measurements == NULL) exit(1);
    for(i=0; i<n; i++)
        scanf("%f", &measurements[i]);

    for(i=0; i<n; i++)
        mo += measurements[i];

    mo = mo / n;
    free (measurements);
    ...
}
```

- ❖ Η συνάρτηση malloc:

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

- ❖ Η malloc δεσμεύει μνήμη δοσμένου μεγέθους και επιστρέφει μια διεύθυνση
- ❖ Όρισμα = το μέγεθος της αιτούμενης μνήμης (# bytes)
 - π.χ., στην πράξη θα χρειαστούμε μνήμη για αποθήκευση int, μνήμη για αποθήκευση float, μνήμη για αποθήκευση double,...
 - **Το μέγεθος της μνήμης δίνεται ως πολλαπλάσιο των sizeof(int), sizeof(float), sizeof(double),...**

Δυναμική δέσμευση μνήμης

- ❖ Η `malloc` δεν μπορεί να επιστρέφει διαφορετικούς τύπους από pointers
 - Επιστρέφει έναν «γενικό» τύπο pointer:
`void *malloc(int size);`
- ❖ Επομένως, είναι πάντα απαραίτητο να χρησιμοποιούμε το κατάλληλο casting, π.χ. (`int *`), για αυτό που επιστρέφει η `malloc`.
- ❖ Άρα αν για ένα μονοδιάστατο πίνακα τύπου `<T>` (όπου `<T>` είναι `int`, `float`, `double`, `char...`) δεν ξέρω τη διάσταση του πριν την εκτέλεση του προγράμματος,
- ❖ τότε ορίζω δείκτη `<T*>` `p` και χρησιμοποιώ την `malloc`
`p = (<T*>) malloc(N*sizeof(<T>));`
Π.χ.
`float *p;`
`p = (float *) malloc(N*sizeof(float));`

Παρένθεση - υπενθύμιση

- ❖ Το α και β παρακάτω είναι ίδια; Αν όχι υπάρχει κάποιο πρόβλημα;

(a)	(b)
<code>int x, *y = &x;</code>	<code>int x, *y; *y = &x;</code>

- ❖ Θυμηθείτε ότι άλλο σημαίνουν οι τελεστές της C μέσα σε μία ΔΗΛΩΣΗ και άλλο μέσα σε μία πράξη. Το (b) λοιπόν είναι διαφορετικό (και λάθος). Το (a) θα ήταν ισοδύναμο με το (c):

(a)	(c)
<code>int x, *y = &x;</code>	<code>int x, *y; y = &x;</code>

- ❖ Επομένως, η σωστή αρχικοποίηση ενός δείκτη με `malloc()` είναι μία από τις δύο παρακάτω:

(a)	(b)
<code>int *y; y = (int *) malloc(40);</code>	<code>int *y = (int *) malloc(40);</code>

- ❖ Η μνήμη που δεσμεύουμε μέσω της `malloc()`:
 - Δεν είναι αρχικοποιημένη (περιέχει τυχαίες τιμές)
 - Αποδεσμεύεται αυτόματα στο τέλος του προγράμματος και επιστρέφεται στο λειτουργικό σύστημα
 - Επομένως, όσο ζητάμε νέους χώρους μνήμης, τόσο μειώνεται η ελεύθερη μνήμη του συστήματος (θέλει προσοχή ώστε να μην φτάσουμε σε “out of memory”)
- ❖ Μπορούμε να αποδεσμεύσουμε / απελευθερώσουμε μνήμη με τη συνάρτηση `free`:

```
void free(void *ptr);
```

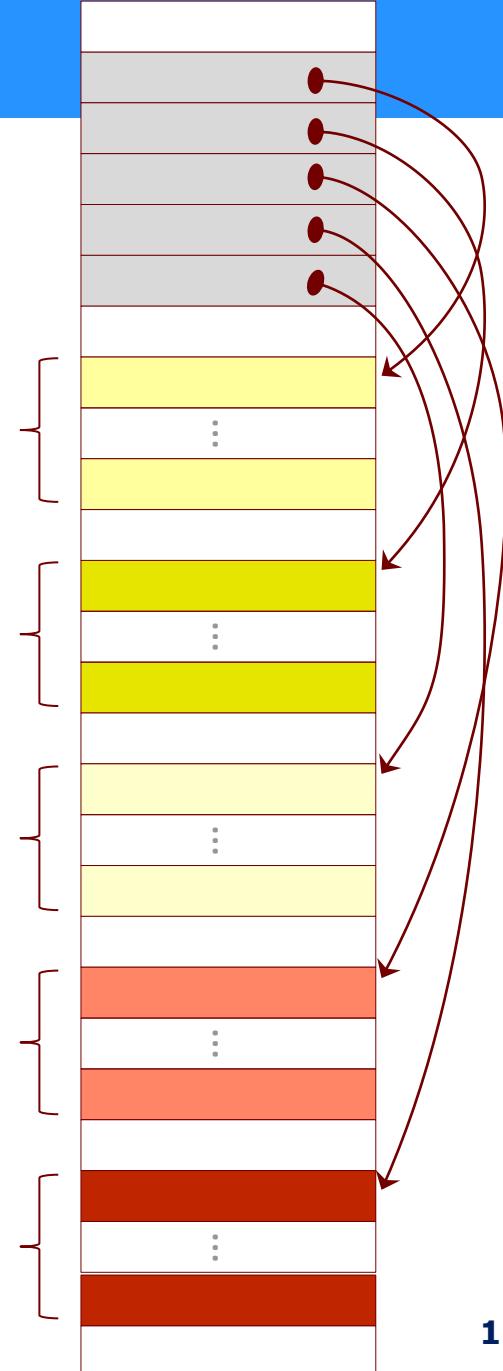
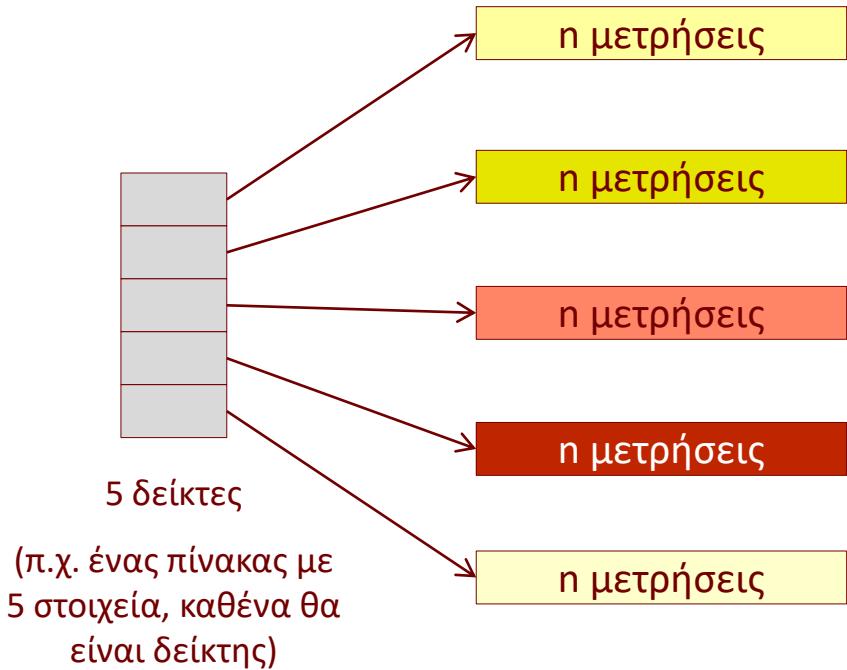
όπου ο `ptr` δείχνει στην αρχή του χώρου που πριν τον είχαμε δεσμέυσει με `malloc()`.

5 πειράματα με η μετρήσεις

- Κατασκευάστε ένα πρόγραμμα το οποίο δέχεται ως είσοδο από το χρήστη τον συνολικό αριθμό μετρήσεων που έγιναν κατά τη διάρκεια **5 πειραμάτων**
- εν συνεχεία ο χρήστης εισάγει στο πρόγραμμα τις επιμέρους τιμές των μετρήσεων για κάθε πείραμα
- τέλος το πρόγραμμα επεξεργάζεται τα δεδομένα
 - μεταξύ άλλων υπολογίζει το μέσο όρο των μετρήσεων



Σχέδιο μνήμης



Δυναμική δέσμευση μνήμης & δείκτες

```
#include <stdio.h>
#include <stdlib.h>
#define M 5

int main() {
    int n, i,j; float mo;
    float *measurements[M]; /* Πίνακας από M δείκτες / ίδιο με float *(measurements[M]); */

    scanf("%d", &n);
    for(i=0; i<M; i++){ /* Για κάθε πείραμα, μνήμη για n μετρήσεις */
        measurements[i] = (float *) malloc(n*sizeof(float)); /* Δείχνει σε χώρο n αριθμών */
        if (measurements[i] == NULL) return 1;
    }

    for(i=0; i<M; i++)
        for(j=0; j<n; j++)
            scanf("%f", &measurements[i][j]); /* Χειρισμός σαν να είναι διδιάστατος πίνακας */
    for(i=0; i<M; i++) {
        for(j=0,mo=0.0; j<n; j++)
            mo += measurements[i][j];
        mo = mo / n;
        printf("%f\n", mo);
    }

    for(i=0; i<M; i++)
        free (measurements[i]); /* Απελευθέρωση μνήμης */
    return 0;
}
```

Ερωτήσεις κρίσεως στην προηγούμενη διαφάνεια

- ❖ Πώς μπορώ να γράψω την παρακάτω έκφραση κάνοντας χρήση μόνο δεικτών και πράξεων (όχι αγκυλών)

```
if (measurements[i] == NULL) return;
```

- ❖ Έτσι:

```
if (*measurements+i) == NULL) return;
```

- ❖ Το παρακάτω;

```
mo = measurements[i][j];
```

- ❖ Έτσι:

```
mo = *( *(measurements+i) + j );
```

- ❖ Διότι είναι ισοδύναμο με:

```
mo = *( measurements[i] + j );
```

- ❖ το οποίο είναι ισοδύναμο με:

```
mo = ( measurements[i] )[j];
```

Πολλά πειράματα, καθένα πολλές μετρήσεις

(Άγνωστος ο αριθμός των πειραμάτων εκ των προτέρων)

- ❖ Κατασκευάστε ένα πρόγραμμα το οποίο δέχεται ως είσοδο από τον χρήστη
 - τον συνολικό αριθμό πειραμάτων που έγιναν
 - τον συνολικό αριθμό μετρήσεων που έγιναν κατά τη διάρκεια ενός πειράματος
- ❖ εν συνεχεία ο χρήστης εισάγει στο πρόγραμμα τις επιμέρους τιμές των μετρήσεων για κάθε πείραμα
- ❖ τέλος το πρόγραμμα επεξεργάζεται τα δεδομένα
 - μεταξύ άλλων υπολογίζει το μέσο όρο των μετρήσεων

Δυναμική δέσμευση μνήμης & δείκτες

- ❖ Στο προηγούμενο πρόβλημα γνωρίζαμε των αριθμό των πειραμάτων (`#define M 5`) και με βάση τον αριθμό των μετρήσεων (`int n`) κατασκευάσαμε έναν πίνακα από δείκτες, καθένας εκ των οποίων έδειχνε σε μια «γραμμή» από n μετρήσεις
 - Το χειριζόμαστε σαν να είναι διδιάστατος πίνακας `measurements[5][n]`
- ❖ Στο πρόβλημα αυτό πρέπει να κατασκευάσουμε δυναμικά και τον πίνακα από τους δείκτες, μιας και τόσο το πλήθος των μετρήσεων ανά πείραμα (n) όσο και το πλήθος των πειραμάτων (m) είναι μεταβλητές του προγράμματος.

Παράδειγμα

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

int main() {
    int n,m;
    int i,j; float mo;
float **measurements;

    scanf("%d", &m);      /* Μαθαίνουμε τις διαστάσεις */
    scanf("%d", &n);

    /* Μνήμη για τη δείκτες σε πειράματα */
measurements = (float **) malloc( m*sizeof(float *) );
    if(measurements == NULL)
        exit(1);

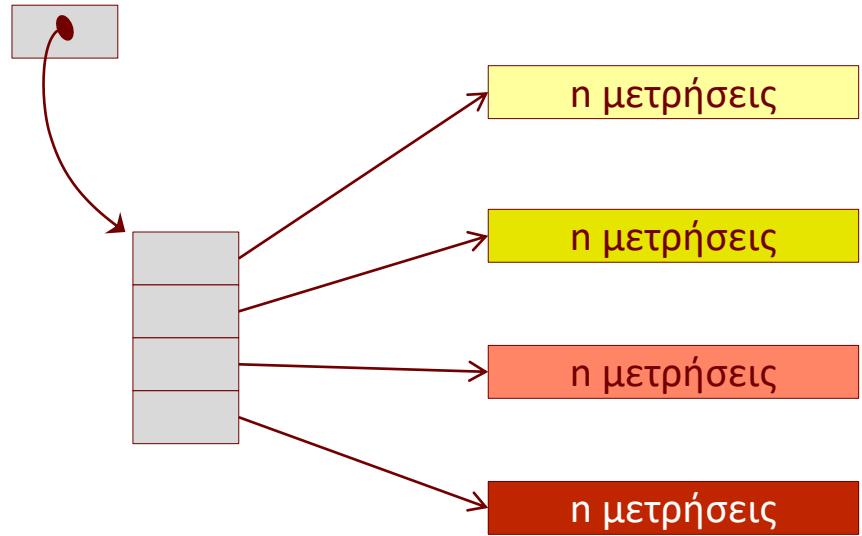
    for(i=0; i<m; i++) { /* Για κάθε πείραμα, μνήμη για n μετρήσεις */
        measurements[i] = (float *) malloc(n*sizeof(float));
        if (measurements[i] == NULL)
            exit(1);
    }
}
```

Παράδειγμα

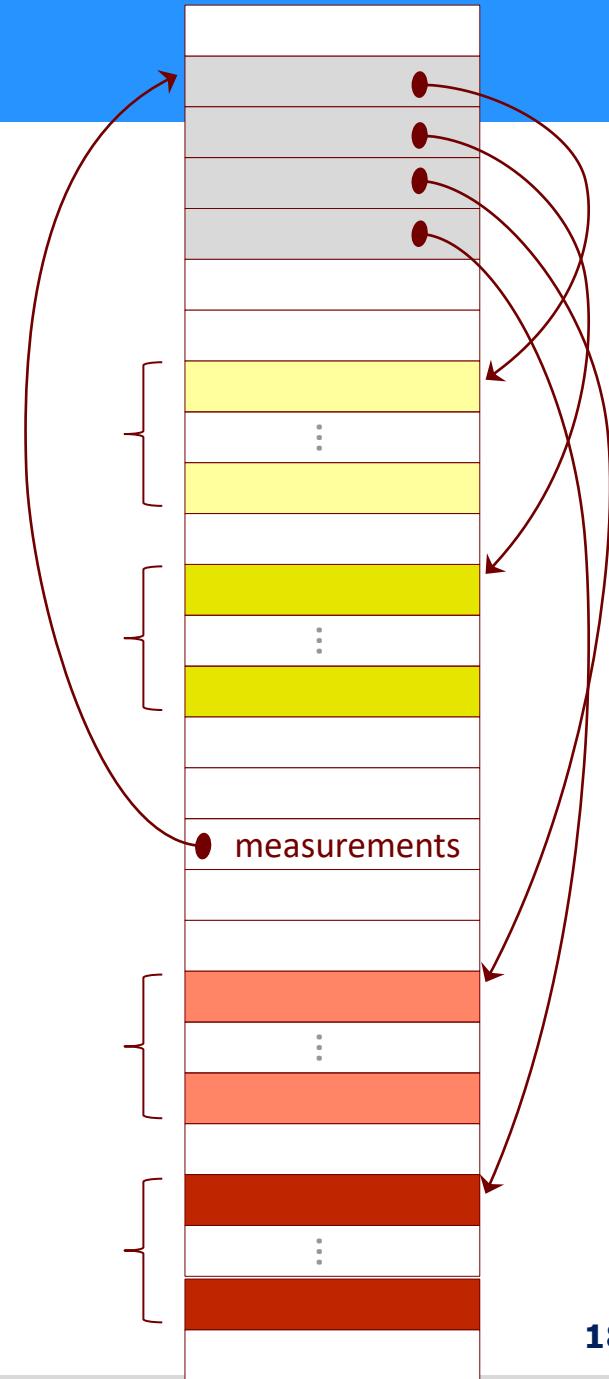
```
for(i=0; i<m; i++)  
    for(j=0; j<n; j++) {  
        scanf("%f", &measurements[i][j]);  
    }  
  
for(i=0; i<m; i++) {  
    for(j=0, mo=0.0; j<n; j++)  
        mo += measurements[i][j];  
    mo = mo / n;  
    printf("%f\n", mo);  
}  
  
for(i=0; i<m; i++)  
    free(measurements[i]); /* Απελευθέρωση γραμμής i */  
free(measurements); /* Απελευθέρωση του πίνακα δεικτών */  
  
return 0;  
}
```

Σχεδιάγραμμα

Δείκτης σε δυναμικό πίνακα

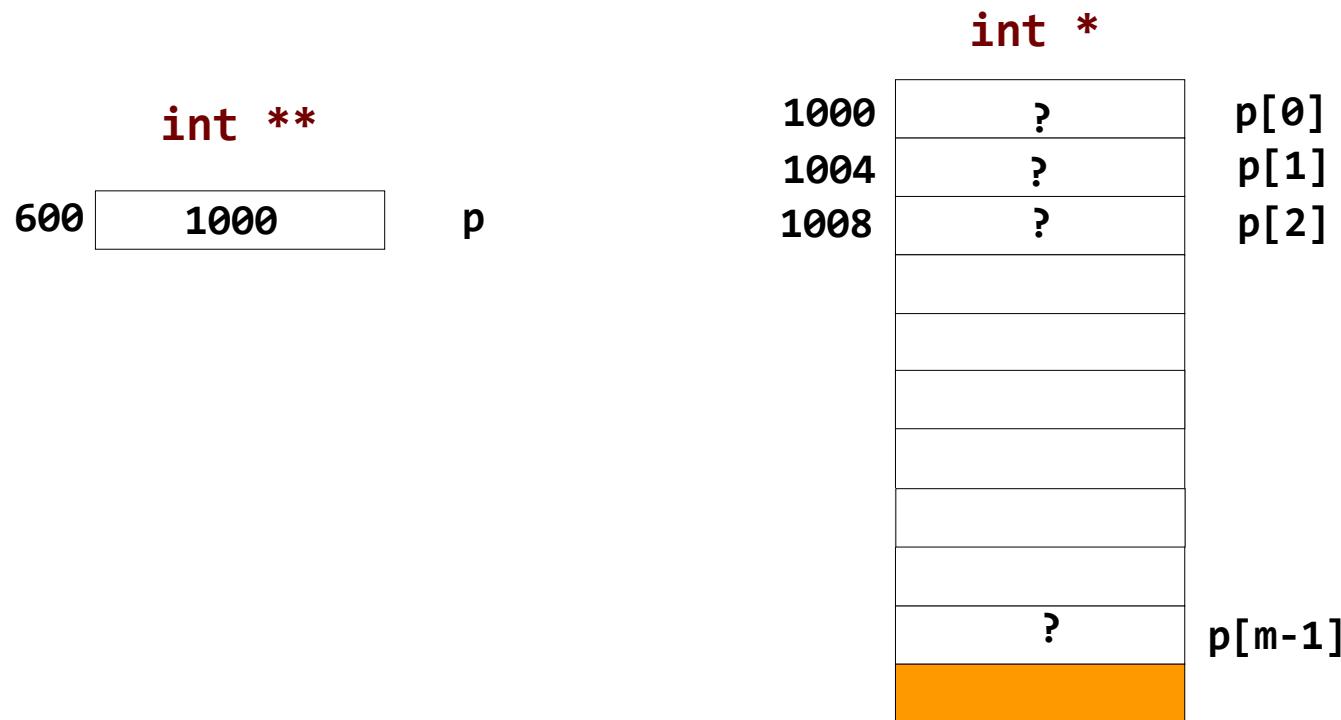


πίνακας με $m=4$
στοιχεία, καθένα θα
είναι δείκτης



Δυναμική δέσμευση και πίνακες δεικτών

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

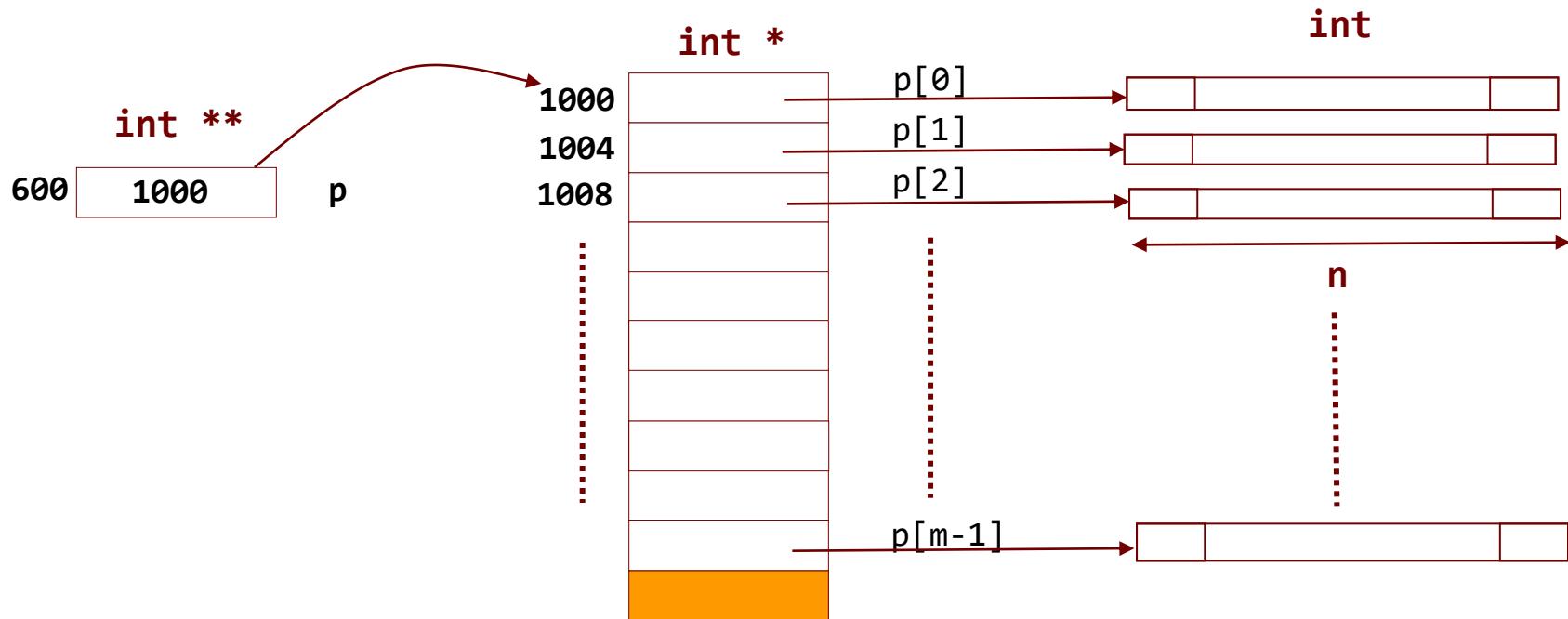


Δυναμική δέσμευση και πίνακες δεικτών

- ❖ Αν στη συνέχεια καλέσω

```
p[i] = (int *) malloc(n*sizeof(int));
```

- ❖ Έτσι έχω δημιουργήσει κάτι σαν «διδιάστατο» πίνακα (mxn), όπως τους φτιάχνει και η Java



Προηγούμενο παράδειγμα – κλήση σε συνάρτηση

```
for(i=0; i<m; i++)  
    for(j=0; j<n; j++) {  
        scanf("%f", &measurements[i][j]);  
    }  
  
show_array(m, n, measurements); /* Πώς ορίζεται; */  
  
for(i=0; i<m; i++) {  
    for(j=0, mo=0.0; j<n; j++)  
        mo += measurements[i][j];  
    mo = mo / n;  
    printf("%f\n", mo);  
}  
  
for(i=0; i<m; i++)  
    free(measurements[i]);           /* Απελευθέρωση γραμμής i */  
free(measurements);                 /* Απελευθέρωση του πίνακα δεικτών */  
  
return 0;  
}
```

Παράμετροι σε συνάρτηση

```
void show_array(int r, int c, ?????)
    int i, j;
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++)
            printf("%f\n", arr[i][j]);
        printf("\n");
    }
}
```

```
void show_array(int r, int c, float arr[r][c])
```

- Στη C δεν επιτρέπονται πίνακες μεταβλητού μεγέθους

```
void show_array(int r, int c, float arr[][])
```

- Και δεν επιτρέπεται αλλά και δεν γνωρίζει το μέγεθος των γραμμών

```
void show_array(int r, int c, float *arr[])
```

- Μια χαρά! Πίνακας από pointers!

```
void show_array(int r, int c, float **arr)
```

- Μια χαρά! Pointer που δείχνει σε pointers (δηλ. στον πρώτο pointer ενός πίνακα από pointers).

```
void show_array(int r, int c, float (*arr)[])
```

- Λάθος! Pointer σε πίνακα από floats.



A) Δημιουργία **τριγωνικού** πίνακα με 10 γραμμές.

- Δηλαδή η γραμμή 0 θα έχει 10 στοιχεία, η γραμμή 1 θα έχει 9 στοιχεία, κλπ, και η γραμμή 9 θα έχει 1 στοιχείο.



B) Δημιουργία τριγωνικού πίνακα με το χρήστη να καθορίζει
κατά το χρόνο εκτέλεσης τον αριθμό των γραμμών

Τριγωνικός πίνακας 10x10 (I)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i, j, *a[10]; /* Πίνακας με 10 δείκτες, σε 10 γραμμές */

    for (i = 0; i < 10; i++)
        a[i] = (int *)malloc((10-i)*sizeof(int)); /* Γραμμή i */

    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10-i; j++) {
            a[i][j] = i + j;
            printf("a(%d,%d)=%d ", i, j, a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Τριγωνικός πίνακας 10xN (II)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i, j, int *a[10], N;

    scanf("%d", &N);
    for (i = 0; i < 10; i++)
        a[i] = (int *)malloc((N-i)*sizeof(int));

    for (i = 0; i < 10; i++) {
        for (j = 0; j < N-i; j++) {
            a[i][j] = i + j;
            printf("a(%d,%d)=%d ", i, j, a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Τριγωνικός πίνακας NxN

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i, j, **a, N;
    scanf("%d", &N);
    a = (int **) malloc(N*sizeof(int *));
    for (i = 0; i < N; i++)
        a[i] = (int *)malloc((N-i)*sizeof(int));

    for (i = 0; i < N; i++) {
        for (j = 0; j < N-i; j++) {
            a[i][j] = i + j;
            printf("a(%d,%d)=%d ", i, j, a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Πώς δεσμεύω καθαρά διδιάστατους πίνακες;

- ❖ Δηλαδή όχι απλά πίνακα από pointers που δείχνουν σε Μ σκόρπιες γραμμές (τον οποίο τον χειρίζομαι μεν σαν να είναι διδιάστατος, όμως δεν καταλαμβάνει συνεχόμενο χώρο στη μνήμη)
- ❖ **αλλά** ένα πίνακα που όντως καταλαμβάνει συνεχόμενο χώρο $M*N$ στοιχείων στην μνήμη;

Απάντηση:

- ❖ Δεν γίνεται – η `malloc()` επιστρέφει πάντα ένα συνεχόμενο γραμμικό χώρο.
- ❖ Μπορώ όμως να τον χειριστώ εγώ «με το χέρι».
- ❖ Αν δεσμεύσω `a = (int *) malloc(M*N*sizeof(int));`; ποιο είναι το στοιχείο στην i γραμμή και j στήλη;
 - Το `a[i*N + j]`.

Χειρισμός διδιάστατου πίνακα με malloc()

```
void func() {  
    int i, j, M, N;  
    int *a;  
  
    scanf("%d", &M); /* Γραμμές */  
    scanf("%d", &N); /* Στήλες */  
    a = (int *) malloc(M*N*sizeof(int));  
  
    for (i = 0; i < M; i++)  
        for (j = 0; j < N; j++)  
            a[i*N+j] = 1;  
  
    free(a); /* 1 free για όλο το χώρο */  
}
```

Χειρισμός διδιάστατου πίνακα με malloc() - ευκολότερα

```
void func() {  
    int i, j, M, N;  
    int *a;  
    #define mat(r,c) a[r*N+c]      /* Preprocessor macro */  
  
    scanf("%d", &M);                /* Γραμμές */  
    scanf("%d", &N);                /* Στήλες */  
    a = (int *) malloc(M*N*sizeof(int));  
  
    for (i = 0; i < M; i++)  
        for (j = 0; j < N; j++)  
            mat(i,j) = 1;           /* Replaced by a[i*N+j] */  
  
    free(a);                      /* 1 free για όλο το χώρο */  
}  
  
/* Το macro δεν είναι τέλειο - θα τα δούμε αργότερα */
```

Επιλογή τύπου πίνακα 2D

- ❖ Όταν δεν μου ζητείται ρητά ο ένας τύπος ή ο άλλος (δηλαδή καθαρά διδιάστατος ή «γιαλαντζί» διδιάστατος – πίνακας από δείκτες σε σκόρπιες γραμμές), τι κάνω;
- ❖ Εξαρτάται από την εφαρμογή.
 - Π.χ. αν καλείται κάποια συνάρτηση που περιμένει ως παράμετρο έναν (πραγματικό) διδιάστατο τότε είμαι **αναγκασμένος** να δεσμεύσω χώρο για έναν τέτοιον πίνακα και να τον χειριστώ όπως δείξαμε.
 - Π.χ. αν οι γραμμές δεν έχουν ίδιο μέγεθος, αναγκαστικά θα πρέπει να έχω τον δεύτερο τύπο.
 - Αν δεν υπάρχει περιορισμός, είναι συνήθως πιο καλό / εύκολο / ευέλικτο να έχω έναν πίνακα από pointers σε σκόρπιες γραμμές,
 - παρά το γεγονός ότι ένας τέτοιος πίνακας έχει λίγο παραπάνω κόπο στο να δημιουργηθεί (πρέπει να κάνω `malloc` κάθε γραμμή ξεχωριστά) και να ελευθερωθεί (πρέπει να κάνω `free` κάθε γραμμή ξεχωριστά).

Ταξινόμηση strings (το πλήθος δίνεται ως όρισμα στη main) |

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void strBubbleSort(char *strings[], int num); /* Πρωτότυπο */
#define SIZE 30           /* Μέγιστο μήκος για κάθε string */

int main(int argc, char *argv[]) {
    int i, NumOfStrings;
    char **array;           /* Για αποθήκευση των strings */

    if (argc < 2) return 1;      /* ή exit(1) */
    NumOfStrings = atoi(argv[1]);
    if (NumOfStrings < 1) return;

    /* Μνήμη για τον πίνακα */
    array = (char **) malloc(NumOfStrings*sizeof(char *));
    if (array == NULL) {
        printf("no memory!\n");
        return;
    }
    /* Διάβασμα των string */
    for (i = 0; i < NumOfStrings; i++) { /* mallocs & αντιγραφή strings */
        array[i] = (char *) malloc(SIZE*sizeof(char));
        if (array[i] == NULL) {
            printf("no memory!\n");
            return;
        }
        fgets(array[i], 29, stdin);
    }
}
```

(Συνέχεια στην επόμενη διαφάνεια)

Ταξινόμηση strings (το πλήθος δίνεται ως όρισμα στη main) II

(Συνέχεια της main())

```
strBubbleSort(array, NumOfStrings);      /* Ταξινόμηση */
puts("\n\nThe sorted list in ascending order is:");
for (i = 0; i < NumOfStrings; i++)
    puts(array[i]);
}

/* Προσέξτε ότι δεν μετακινούνται τα strings - μόνο οι pointers
 */
void strBubbleSort(char *strings[], int num) {
    char *temp;
    int top, seek;

    for (top = 0; top < num-1; top++) {
        for (seek = top+1; seek < num; seek++) {
            if (strcmp(strings[top], strings[seek]) > 0) {
                temp = strings[seek];           /* Εναλλαγή pointers */
                strings[seek]= strings[top];
                strings[top] = temp;
            }
        }
    }
}
```

Άλλες κλήσεις διαχείρισης μνήμης

```
void *calloc(size_t n, size_t size);
```

- Δεσμεύει χώρο στον οποίο αρχικοποιεί όλα τα bytes σε μηδέν
- Παίρνει 2 παραμέτρους: το πλήθος των στοιχείων και το μέγεθος του κάθε στοιχείου (σε bytes):
 - ✧ `int *p = (int *) calloc(n, sizeof(int));`

```
void *realloc(void *p, size_t size);
```

- Δεσμεύει νέο χώρο και ότι υπήρχε στον παλιό (όπου δείχνει ο p) αντιγράφεται στον νέο.
- Αν το νέο size είναι μικρότερο τότε χάνονται κάποια δεδομένα που υπήρχαν στον παλιό (p)

Παράδειγμα

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

int main() {
    int *p, *q;
    p = (int *) malloc(10*sizeof(int));
    q = (int *) realloc(p, 20*sizeof(int));
    printf("%p, %p\n", p, q);      /* μπορεί να διαφέρουν */
    return 0;
}
```

Αποδέσμευση μνήμης

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

Π.χ.

```
p = (int *) malloc(10*sizeof(int)); /* Δέσμευση */  
...  
free(p); /* Αποδέσμευση */
```

- ❖ Αποδεσμεύει τον χώρο στον οποίον δείχνει ο p. Ο χώρος αυτός πρέπει να έχει προέλθει από malloc/calloc/realloc.
- ❖ Προσοχή:
 - Ο χώρος αποδεσμεύεται (επιστρέφεται στο σύστημα και δεν μπορούμε να τον ξαναχρησιμοποιήσουμε), ΑΛΛΑ
 - ο pointer p ΣΥΝΕΧΙΖΕΙ ΝΑ ΔΕΙΧΝΕΙ ΕΚΕΙ (δεν άλλαξε η τιμή του)!!

- ❖ `#include <stdlib.h>`
- ❖ `void *malloc(size_t size);`
Δέσμευση μνήμης
- ❖ `void *calloc(size_t count, size_t size);`
Δέσμευση μνήμης αρχικοποιημένης σε μηδενικά bytes
- ❖ `void *realloc(void *ptr, size_t size);`
Επαναδέσμευση μνήμης (επέκταση/συρρίκνωση)
- ❖ `void free(void *ptr);`
Αποδέσμευση μνήμης



- ❖

```
int *p;
p = (int *) malloc(sizeof(int));
```
- ❖

```
int *p;
p = (int *) malloc(M*sizeof(int));
```
- ❖

```
int *p[10];
for (i = 0; i < 10; i++)
    p[i] = (int *) malloc(M*sizeof(int));
```
- ❖

```
int **p;
p = (int **) malloc(10*sizeof(int *));
for (i = 0; i < 10; i++)
    p[i] = (int *) malloc(M*sizeof(int));
```

Pointer (malloc) από συνάρτηση |

```
#include <stdio.h>

void makeArray(int * ip, int size) {
    int i;
    ip = (int *) malloc(size*sizeof(int));
    for (i=0; i<size; i++){
        ip[i] = 3*i+45;
        printf("print from mA: %d\n",ip[i]);
    }
}
int main() {
    int * array = NULL;
    int i, N=12;

    makeArray(array, N);
    for (i=0;i<N;i++)
        printf("print from main: %d\n",array[i]);
    return 0;
}
```

ΔΟΥΛΕΥΕΙ ????

Pointer (malloc) από συνάρτηση II

```
#include <stdio.h>

void makeArray(int **ip, int size) {
    int i;
    *ip = (int *) malloc(size*sizeof(int));
    for (i=0; i<size; i++){
        (*ip)[i] = 3*i+45;      //try without ()
        printf("print from mA: %d\n",*((*ip)+i)); //aka (*ip)[i]
    }
}
int main() {
    int * array = NULL;
    int i, N=12;

    makeArray(&array, N);
    for (i=0;i<N;i++)
        printf("print from main: %d\n",array[i]);
    return 0;
}
```

ΔΟΥΛΕΥΕΙ !!!!!

Προβλήματα pointers / διαχ. μνήμης – Dangling pointers

```
int *p;  
printf("%d", *p);
```

Το p δεν δείχνει σε καμία θέση (σκουπίδια) - **dangling**

```
int *p;  
if (condition) {  
    int temp = 1;  
    p = &temp;  
    printf("%d", *p);  
}  
printf("%d", *p);
```

Μετά το μπλοκ του if {}, το temp ΔΕΝ ΥΦΙΣΤΑΤΑΙ. Επομένως το p δείχνει σε άκυρη / μη νόμιμη θέση - **dangling**

```
int *p;  
p = malloc(10*sizeof(int));  
p[5] = 100;  
...  
free(p);  
...  
p[5]--;
```

Μετά το free(), η δεσμευμένη μνήμη από το malloc() ΔΕΝ ΥΠΑΡΧΕΙ. Όμως, το p ΔΕΝ ΕΧΕΙ ΑΛΛΑΞΕΙ! Επομένως δείχνει σε άκυρη θέση - **dangling**

Κίνδυνοι από dangling pointers & προφυλάξεις

- ❖ Οι dangling pointers μπορεί να οδηγήσουν σε προσπέλαση άκυρων / παράνομων διευθύνσεων.
 - Ένα πρόβλημα που μπορούν να προκαλέσουν είναι το κρασάρισμα ή χωρίς αιτιολογία δυσλειτουργία του προγράμματος (εξαιρετικά δύσκολο debugging)
 - Ακόμα πιο σοβαρό είναι η πρόκληση ζημιάς / κρασάρισμα ΟΛΟΥ του συστήματος.

❖ Προφυλάξεις:

1. Πάντα αρχικοποιούμε τους *pointers* κατά τη δήλωσή τους (τουλάχιστον τους θέτουμε ίσους με *NULL*).
2. Μετά από *to free(p)*, θέτουμε τον *p = NULL*.

Έτσι τουλάχιστον αποφεύγουμε ζημιά στο σύστημα αλλά και διευκολύνουμε το debugging μιας και η προσπέλαση σε *NULL* «χτυπάει» αμέσως.



Προβλήματα pointers / διαχ. μνήμης – Memory leaks

```
char *p;  
p = malloc(10*sizeof(char));  
scanf("%s", p);  
for ( ; p != '\0'; p++ )  
    printf("%c", *p);  
...
```

```
int *p, *q;  
p = malloc(10*sizeof(int));  
q = malloc(20*sizeof(int));  
...  
if (condition) {  
    p = q;  
}  
printf("%d", p[5]);  
...
```

Κανένα πρόβλημα ορθότητας.

Απλά πλέον ο χώρος των 10 θέσεων που δεσμεύσαμε μέσω του p ΔΕΝ ΜΠΟΡΕΙ ΝΑ ΠΡΟΣΠΕΛΑΣΤΕΙ ΜΕ ΤΙΠΟΤΕ.

Ενώ συνεχίζει να υπάρχει στη μνήμη (δεν έχει αποδεσμευθεί), έχει χαθεί ο μοναδικός pointer που γνώριζε τη διεύθυνσή του – **memory leak (διαρροή μνήμης)**

ΜΗΝ ΞΕΧΝΑΤΕ ΤΑ free() !!!

Αναλογία με αποθήκη προϊόντων

- ❖ Γνωστό κατάστημα διαθέτει αποθήκη με πολλά ράφια όπου σε κάθε ράφι υπάρχει ένα προϊόν (έπιπλο). Τα προϊόντα είναι συσκευασμένα έτσι ώστε ΔΕΝ ΜΠΟΡΕΙΣ να καταλάβεις τι είναι αν πας στα ράφια της αποθήκης. Για να πάρεις το έπιπλο πρέπει να επισκεφτείς την έκθεση του καταστήματος και να σημειώσεις σε ειδικά χαρτάκια το ΡΑΦΙ της αποθήκης που έχει το προϊόν που σε ενδιαφέρει. Με το συμπληρωμένο χαρτάκι πας στο σωστό ράφι και το παίρνεις – αλλιώς δεν μπορείς να βρεις το προϊόν.
- ❖ Αναλογία:
 - **Μνήμη** = αποθήκη
 - **Μεταβλητή** (κελί στη μνήμη) = ράφι
 - **Τιμή μεταβλητής** = προϊόν στο ράφι
 - **Δείκτης** = το χαρτάκι

Έννοιες & αναλογίες

Λειτουργία / έννοια	Αναλογία
<code>p = &x;</code>	Γράφω σε χαρτάκι το ράφι.
<code>*p</code>	Το προϊόν που υπάρχει στο ράφι (πάω εκεί και το βρίσκω).
<code>q = &y;</code>	Άλλο χαρτάκι που γράφει άλλο ράφι.
<code>p = q;</code>	Στο πρώτο χαρτάκι αλλάζω τι έγραψα και γράφω το ίδιο ράφι που γράφει το δεύτερο χαρτάκι.
<code>p++;</code>	Αλλάζω το χαρτάκι και σημειώνω το αμέσως επόμενο ράφι.
<code>temp = *a; *a = *b; *b = *a;</code>	Πάω στα ράφια που γράφουν τα χαρτάκια <code>a</code> και <code>b</code> και εναλλάσσω τα προϊόντα.
<code>t = malloc(10*sizeof(int));</code>	Δεσμεύω 10 συνεχόμενα άδεια ράφια στην αποθήκη. Στο χαρτάκι σημειώνω το ΠΡΩΤΟ από αυτά.
Διαρροή μνήμης	Σβήνω το μοναδικό χαρτάκι που έγραφε τα νέα ράφια. Δεν ξέρω πλέον που είναι τα νέα ράφια στην αποθήκη
<code>free(t);</code>	Αποδεσμεύω τα ράφια που δέσμευσα.
<code>v = malloc(10*sizeof(int)); w = v;</code>	Δεσμεύω 10 συνεχόμενα άδεια ράφια στην αποθήκη. Στο χαρτάκι σημειώνω το ΠΡΩΤΟ από αυτά και φτιάχνω και δεύτερο χαρτάκι με το ίδιο ράφι.
<code>free(v); v = NULL;</code>	Αποδεσμεύω τα ράφια που δέσμευσα. Σβήνω το πρώτο χαρτάκι.
Dangling pointer.	Το δεύτερο χαρτάκι (<code>w</code>) συνεχίζει να γράφει το πρώτο ράφι από τα αποδεμευμένα.

Η γλώσσα C

*Νέοι και σύνθετοι τύποι
(typedef & structs & unions)*



MYY502

typedef (I)

- ❖ Η C επιτρέπει να δώσουμε **οποιαδήποτε ονομασία μας αρέσει σε έναν υπάρχων τύπο**.
- ❖ Π.χ. δεν μου αρέσει το “int”, “char” κλπ.
 - Μπορώ να τα «βαφτίσω» με δεύτερο όνομα (ψευδώνυμο) με την typedef.

- ❖ Παραδείγματα:

```
typedef int Integer;      /* ψευδώνυμο του τύπου int */
typedef char Character; /* ψευδώνυμο του τύπου char */
int      n, m;
Integer x;
Character c;

x = n;
c = 'a';
```

typedef (II)

- ❖ Η C επιτρέπει να ορίσουμε **έναν νέο τύπο, με όποια ονομασία μας αρέσει.**
- ❖ Παράδειγμα: Θέλω να ορίσω τον τύπο «δείκτη σε χαρακτήρα» και να τον ονομάσω «pchar».
 - Το κάνω πάλι με την **typedef**.

```
typedef char * pchar;           /* Νέος τύπος, “pchar” */
char c, name[20];
pchar s, t;      /* pointer, αλλά χωρίς το αστεράκι!! */
s = &c;
t = name;        /* Δηλαδή, t = &name[0] */
*s = t[1];
```

Πολύπλοκα `typedef`

- ❖ Και πιο πολύπλοκοι τύποι!
- ❖ Π.χ. θέλω να ορίσω τον τύπο intarray, «πίνακας 10 ακεραίων».

```
typedef int intarray[10];
```

```
intarray A = { 0, 1, 2 };
```

```
A[5] = 15;
```

- ❖ Π.χ. τύπος πίνακα με 10 pointers σε χαρακτήρες

```
typedef char * pchar; /* Νέος τύπος, "pchar" */
```

```
typedef pchar pcarray[10]; /* Πίνακας με 10 pchar */
```

```
pcarray names;
```

```
names[0] = "Bill";
```

```
names[1] = "Demi";
```

```
names[2] = names[0];
```



Συνταγή για `typedef`

- ❖ Αν θέλετε να ορίσετε έναν νέο τύπο, αλλά δυσκολεύεστε στο πως να τον εκφράσετε, υπάρχει ο εξής απλός κανόνας:
 1. Σκεφτείτε ένα όνομα (π.χ. `mytype`)
 2. Ορίστε μία μεταβλητή `mytype` ακριβώς όπως θα την θέλατε
 3. Τοποθετήστε τη λέξη “`typedef`” μπροστά από τη δήλωση της μεταβλητής.
- ❖ Παράδειγμα:
 - Κάνω επεξεργασία εικόνων και για κάθε πίξελ χρησιμοποιώ ένα πίνακα 3 χρωμάτων (RGB – red, green, blue). Θέλω να ορίσω έναν νέο τύπο που να μου δίνει αυτόν τον πίνακα.
 1. Θα τον πω “`color`”.
 2. Είναι πίνακας 3 ακεραίων, άρα αν ήταν μεταβλητή θα οριζόταν ως εξής:
`int color[3];`
 3. Ολοκλήρωση με `typedef`:
`typedef int color[3];`
- ❖ Χρήση:
`color c = {50, 50, 50}, pixels[100];`
`pixels[5][0] = c[0];`

Η γλώσσα C

Δομές (*structs*)



MYY502

Εισαγωγή

- ❖ Οι δομές στη C επιτρέπουν τον ορισμό **σύνθετων τύπων δεδομένων** πέραν των βασικών τύπων που προσφέρει η γλώσσα
 - Με τον τρόπο αυτό επιτυγχάνουμε καλύτερη οργάνωση των δεδομένων
- ❖ Πολλές φορές θέλουμε να διατηρούμε συλλογή από πληροφορίες για **ΜΙΑ** οντότητα, π.χ. μία μεταβλητή «εργαζόμενος» θα θέλαμε να εμπεριέχει και το όνομα και το επώνυμο και την ηλικία του, αντί να ορίσουμε 3 ξεχωριστές μεταβλητές.
 - Μια δομή περιλαμβάνει μια συλλογή από **ανομοιογενή πεδία (διαφορετικών τύπων)** που ομαδοποιούνται με ένα μόνο όνομα που αντιστοιχεί στον νέο τύπο δεδομένων
- ❖ Ο ορισμός μιας **μεταβλητής** του νέου τύπου δεδομένων **αντιστοιχεί στον ορισμό μιας συλλογής από μεταβλητές** που αντιστοιχούν με τη σειρά τους **στα πεδία της δομής** που ορίσαμε.

```
struct <ετικέτα δομής> {  
    <δηλώσεις πεδίων>  
};
```

```
struct person {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
};
```

Δηλώσεις μεταβλητών

❖ Προσοχή στη λέξη **struct**:

- Αρχικά χρησιμοποιείται **για να ορίσει την δομή** και την ετικέτα/όνομα της και
- Στη συνέχεια χρησιμοποιείται **για να ορίσει μεταβλητές** αυτού του τύπου δομής.

❖ Παράδειγμα:

```
struct person {      /* Ορισμός ονόματος & πεδίων δομής */
    char firstName[20];
    char lastName[20];
    char gender;
    int age;
}
struct person x;      /* Ορισμός μεταβλητής τύπου struct person */
```

❖ Μπορεί κάποιος να κάνει και τα δύο σε ένα (όχι καλό!):

```
struct person {
    ...
} x;
```

❖ Αρχικοποίηση με αγκύλες:

```
struct person x = {"Στέλιος", "Μπέης", 'M', 40};
```

- Όπως και στους πίνακες, μπορεί να γίνει αρχικοποίηση μόνο μέχρι κάποιο πεδίο. Τα επόμενα πεδία αρχικοποιούνται αυτόματα στο 0.

Πίνακες από δομές

- ❖ Πίνακας από δομές:

```
struct person people[40];
```

```
struct person {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
};
```

- ❖ Ορισμός δομής & μαζί δήλωση μεταβλητής:

```
struct person {  
    ...  
} people[40];
```

- ❖ Αρχικοποίηση:

```
struct person people[] = {  
    {"Στέλιος", "M", 'M', 40},  
    {"Κώστας", "A", 'M', 50}  
};
```

Πρόσβαση στα πεδία μίας δομής

❖ Χρήση: <δομή>.<πεδίο>

- x.firstName
- x.age

```
struct person {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
};
```

❖ Π.χ.

```
struct person x = {"Στέλιος", "Μπέης", 'M', 40};  
printf("%c", x.gender);  
x.age ++;
```

❖ Π.χ.

```
struct person people[40];           /* Πίνακας με δομές */  
...  
printf("%c", people[3].gender);     /* Στοιχείο 3 */  
people[4].age ++;                  /* Στοιχείο 4 */
```



Παράδειγμα I

```
struct person {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
};  
  
int main() {  
    struct person x, y;  
  
    strcpy(x.firstName, "rivaldo");  
    strcpy(x.lastName, "unknown");  
    x.gender = 'M';  
    x.age = 37;  
  
    y = x; /* Προσέξτε αυτό! Αντιγράφονται ΌΛΑ ΤΑ BYTES */  
    return 0;  
}
```

Παράδειγμα II

```
struct person {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
};  
  
int main() {  
    struct person x, y;  
  
    scanf("%s", x.firstName);  
    scanf("%s", x.lastname);  
    scanf("%c", &(x.gender));  
    scanf("%d", &(x.age));  
  
    y = x;  
    return 0;  
}
```

Προσοχή στα πεδία που χρησιμοποιείτε!! (I)

```
struct person {  
    char *firstName;           /* Αντί για πίνακες */  
    char *lastName;  
    char gender;  
    int age;  
};  
  
int main() {  
    struct person x, y;  
  
    scanf("%s", x.firstName);  
    scanf("%s", x.lastname);  
    scanf("%c", &(x.gender));  
    scanf("%d", &(x.age));  
  
    y = x;  
    return 0;  
}
```

Είναι όλα OK???

Προσοχή στα πεδία που χρησιμοποιείτε!! (I)

```
struct person {  
    char *firstName;           /* Αντί για πίνακες */  
    char *lastName;  
    char gender;  
    int age;  
};  
  
int main() {  
    struct person x, y;  
  
    x.firstname = (char *) malloc(20); /* Should check for NULL... */  
    x.lastname = (char *) malloc(20); /* Should check for NULL... */  
    scanf("%s", x.firstName);  
    scanf("%s", x.lastname);  
    scanf("%c", &(x.gender));  
    scanf("%d", &(x.age));  
  
    y = x;  
    return 0;  
}
```

QUIZ

Τι γίνεται στο τέλος
με το y???

typedef για ευκολία

- ❖ Μπορούμε να χρησιμοποιήσουμε την `typedef` ώστε κάνουμε τις δηλώσεις μας απλούστερες, π.χ.

```
struct person_s {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
};  
typedef struct person_s person_t;  
  
int main() {  
    person_t p;  
    p.age = 12;  
    return 0;  
}
```

Στυλ προγραμματισμού:

To `struct` και το όνομα του τύπου από το `typedef` πρέπει να συνδέονται με συστηματικό τρόπο (ή ακόμα και συνώνυμο).

Συνήθης τακτική / σύμβαση:

Struct: person ή person_s

Typdef: Person, person_t

Συνδυασμός struct με typedef

Ορισμός struct και τύπου μαζί:

```
typedef struct person_s {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
} person_t;
```

```
int main() {  
    person_t p;  
    p.age = 12;  
    return 0;  
}
```

Κι άλλη (κακή) εκδοχή: μη-ονοματισμένο struct

```
typedef struct {  
    char firstName[20];  
    char lastName[20];  
    char gender;  
    int age;  
} person_t;
```

```
int main() {  
    person_t p;  
    p.age = 12;  
    return 0;  
}
```

Εμφωλευμένες δομές

- ❖ Οι δομές μπορεί να είναι ένθετες, δηλαδή να φωλιάζονται η μια μέσα στην άλλη, δημιουργώντας πιο πολύπλοκες δομές:

```
struct family {  
    struct person father;  
    struct person mother;  
    int     numofchildren;  
    struct person children[5];  
};
```

Εμφωλευμένες δομές – παράδειγμα

```
struct family {  
    struct person     father;  
    struct person     mother;  
    int               numofchildren;  
    struct person    children[5];  
};  
  
int main() {  
    struct person x, y, z;  
    struct family fml;  
  
    fml.numofchildren = 2;  
    strcpy(fml.father.firstName, "Joe");  
    strcpy(fml.children[0].firstName, "Marry");  
    return 0;  
}
```

Πράξεις/λειτουργίες σε δομές

❖ Επιτρεπτές πράξεις/λειτουργίες σε μια δομή είναι:

- η αντιγραφή της
- η απόδοση τιμής σ' αυτήν ως σύνολο
- η εξαγωγή της διεύθυνσής της με &
- η προσπέλαση των μελών της

❖ Επιτρέπονται

- `<struct> = <struct>` (**αντιγράφονται τα πάντα**)
- `&<struct>` (δείκτης στο χώρο που αποθηκεύεται το struct)
- `<struct>.πεδίο`

❖ Ο δομές δεν μπορούν να συγκριθούν, δηλ. δεν επιτρέπεται:

- `<struct> == <struct>`

Πράξεις σε δομές

- ❖ Μεταβλητές τύπου δομής μπορούν να μεταβιβαστούν ως ορίσματα σε συναρτήσεις όπως επίσης και να επιστραφούν ως αποτελέσματα συναρτήσεων

```
struct person inc_age(struct person x) {  
    x.age += 1;  
    return x;  
}  
  
int main() {  
    struct person x1, x2;  
    x1.age = 45;  
    x2 = inc_age(x1); /* Πέρασμα με τιμή (copy) */  
    return 0;  
}
```

Δομές και δείκτες

❖ Παράδειγμα δείκτη σε δομή

```
struct person p;
```

```
struct person *pp;
```

❖ Νόμιμες εκφράσεις:

```
pp = &p;
```

```
printf("%s", (*pp).firstName);
```

❖ Οι εκφράσεις τύπου **(*pp).firstName** απαιτούν πάντα παρενθέσεις

❖ Εναλλακτικά: **pp->firstName**, δηλαδή

(*pp). ≡ pp->

Παράδειγμα - Μεταβίβαση με τιμή

```
#include <stdio.h>
struct person {
    char firstName[20];
    char lastName[20];
    char gender;
    int age;
};

void initPerson(struct person p) {
    strcpy(p.firstName, "xxxxx");
    strcpy(p.lastName, "yyyyy");
    p.gender = 'M'; p.age = 40;
}

int main() {
    struct person q;
    q.age = 0;
    strcpy(q.firstName, "");
    strcpy(q.lastName, "");
    initPerson(q);          /* Με τιμή */
    printf("%s %s %d\n", q.firstName, q.lastName, q.age);
    return 0;
}
```

Παράδειγμα - Μεταβίβαση με αναφορά

```
#include <stdio.h>
struct person {
    char firstName[20];
    char lastName[20];
    char gender;
    int age;
};

void initPerson(struct person *p) {
    strcpy(p->firstName, "xxxxx");
    strcpy(p->lastName, "yyyyy");
    p->gender = 'M'; p->age = 40;
}

int main() {
    struct person q;
    q.age = 0;
    strcpy(q.firstName, "");
    strcpy(q.lastName, "");
    initPerson(&q);           /* Με αναφορά */
    printf("%s %s %d\n", q.firstName, q.lastName, q.age);
    return 0;
}
```

Για μεγάλες δομές, η μεταβίβαση με δείκτη είναι γενικά αποτελεσματικότερη, επίσης χρειάζεται όταν χρειάζεται να κάνουμε **πέρασμα με αναφορά**.

Παράδειγμα 1

```
#include <stdio.h>
struct person {
    char name[20];
    int age;
};

struct person inc_age(struct person x) {
    x.age += 1;
    return x;
}

int main() {
    struct person a = {"Me", 10};
    struct person c;

    c = inc_age(a);
    printf("C:%d A:%d\n", c.age, a.age);
    return 0;
}
```

\$./a.out

C:11 A:10

Παράδειγμα 2 – δομές και δείκτες

```
#include <stdio.h>
#include <string.h>
struct person {
    char name[20];
    int age;
};

struct person inc_age(struct person x) {
    x.age += 1;
    return x;
}

void inc_age_ptr(struct person *x) {
    x->age += 1;
}

int main() {
    struct person a = {"XXX", 10}, b = {"YYY", 20}, c;

    c = inc_age(a);
    inc_age_ptr(&b);
    printf("C:%d A:%d B:%d \n", c.age, a.age, b.age);
    return 0;
}
```

\$./a.out

C:11 A:10 B:21

Παράδειγμα 3 (1/2)

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

struct name{
    char *fname;
    char *lname;
    int letters;      /* Πλήθος γραμμάτων επώνυμου + μικρού ονόματος */
};

void getinfo(struct name *pst) {
    char temp[81];

    printf("First name ? ");
    fgets(temp, 80, stdin);                      /* Διάβασμα */
    pst->fname = (char *) malloc(strlen(temp) + 1); /* Δέσμευση μνήμης */
    if (pst->fname == NULL) exit(1);
    strcpy(pst->fname, temp);                    /* Αντιγραφή */

    printf("Last name ? ");
    fgets(temp, 80, stdin);
    pst->lname = (char *) malloc(strlen(temp) + 1));
    if (pst->fname == NULL) exit(1);
    strcpy(pst->lname, temp);
}
```

Παράδειγμα 3 (2/2)

```
/* Συνέχεια */

void computeLen(struct name *pst) {
    pst->letters = strlen(pst->fname) + strlen(pst->lname);
}

void cleanup(struct name *pst) {
    free(pst->fname);
    free(pst->lname);
}

int main() {
    struct name x;

    getinfo(&x);
    computeLen(&x);
    printf(" %s %s %d\n", x.fname, x.lname, x.letters);
    cleanup(&x);
    return 0;
}
```

Παράδειγμα – πίνακες και δομές (1/2)

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

typedef struct student { /* Θα φυλάμε βαθμολογίες μαθητών */
    int AM;
    int grade;
} Student;

void calculate_stats(Student class[], int studNo);
void print_student_details(Student class[], int studNo);

int main(int argc, char *argv[]) {
    Student *class;
    int i, n;

    n = atoi(argv[1]); /* Εκτέλεση του προγράμματος με ορίσματα */
    class = (Student *) malloc(n*sizeof(Student));
    for (i = 0; i < n; i++) /* Διάβασμα η φοιτητών-βαθμών */
        scanf("%d%d", &class[i].AM, &class[i].grade);

    print_student_details(class, n);
    calculate_stats(class, n);
    return 0;
}
```

Παράδειγμα – πίνακες και δομές (2/2)

```
void calculate_stats(Student class[], int studNo) {  
    int i;  
    int max = 0;  
  
    for (i = 0; i < studNo; i++) {  
        if (class[i].grade > max)  
            max = class[i].grade;  
    }  
    printf("MAX GRADE = %d\n", max);  
}  
  
void print_student_details(Student class[], int studNo) {  
    int i;  
  
    printf("AA \t AM \t GRADE\n");  
  
    for (i = 0; i < studNo; i++) {  
        printf("%d \t %d \t %d\n", i, class[i].AM, class[i].grade);  
    }  
}
```

Σύνθετο παράδειγμα

```
struct Name { char first[20]; char last[20]; };
struct An8rwpos { struct Name gname; float income; }

int main() {
    struct An8rwpos people[2] = { {"A", "B"}, 123.4}, {"C", "D"}, 345.67} ;
    struct An8rwpos *p, me;

    p = &people[0];      /* Δείχνει στο 0 στοιχείο του people */
    me = *(p+1);        /* = p[1] (το πρώτο στοιχείο του p) */
    printf("%s %s %f \n\n", /* A B 123.400000 */
           p->gname.first, p->gname.last, p->income);
    printf("%s %s %f \n", people[1].gname.first,
           people[1].gname.last, people[1].income);
    printf("%s %s %f \n",
           me.gname.first, me.gname.last, me.income);
    printf("%s %s %f \n\n", (*(p+1)).gname.first,
           (*(p+1)).gname.last, (*(p+1)).income);

    strcpy(p[1].gname.first, "X");
    strcpy(p[1].gname.last, "Y");
    p[1].income = 0.0;

    printf("%s %s %f \n", people[1].gname.first,
           people[1].gname.last, people[1].income);
    printf("%s %s %f \n\n",
           me.gname.first, me.gname.last, me.income);
}
```

\$./a.out

A	B	123.400000
C	D	345.670000
C	D	345.670000
C	D	345.670000
X	Y	0.000000
C	D	345.670000

Ισοδύναμες εκφράσεις στο προηγούμενο παράδειγμα

```
/* struct An8rwpos people[2] = { {"A", "B"}, 123.4},  
                                {"C", "D"}, 345.67} };  
  
struct An8rwpos *p, me;  
p = &people[0];  
me = *(p+1);  
*/
```

- ❖ Υπάρχουν δύο σύνολα ισοδύναμων εκφράσεων
- ❖ 1^ο σύνολο:
 - `printf("%s\n", p->gname.first);`
 - `printf("%s\n", people[0].gname.first);`
 - `printf("%s\n", (*p).gname.first);`
- ❖ 2^ο σύνολο:
 - `printf("%s\n", me.gname.first);`
 - `printf("%s\n", people[1].gname.first);`
 - `printf("%s\n", (*(p+1)).gname.first);`

Η γλώσσα C

Αναφορές σε δομές



Αναφορές (προς ίδιου τύπου δομή)

❖ Δεν επιτρέπεται

```
struct Employee {  
    char name[20];  
    int age;  
    struct Employee manager;  
};
```

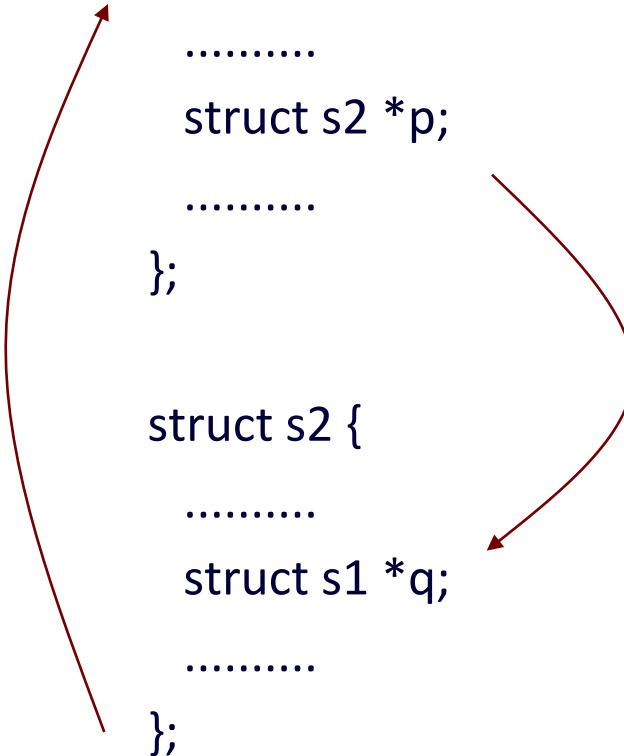
❖ Επιτρέπεται

```
struct Employee {  
    char name[20];  
    int age;  
    struct Employee *manager;  
};
```

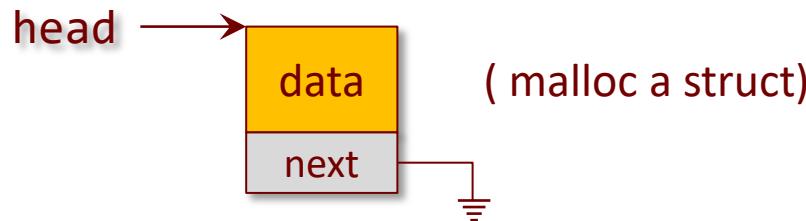
Αναφορές

- ❖ Επίσης επιτρέπεται (κυκλική αναφορά)

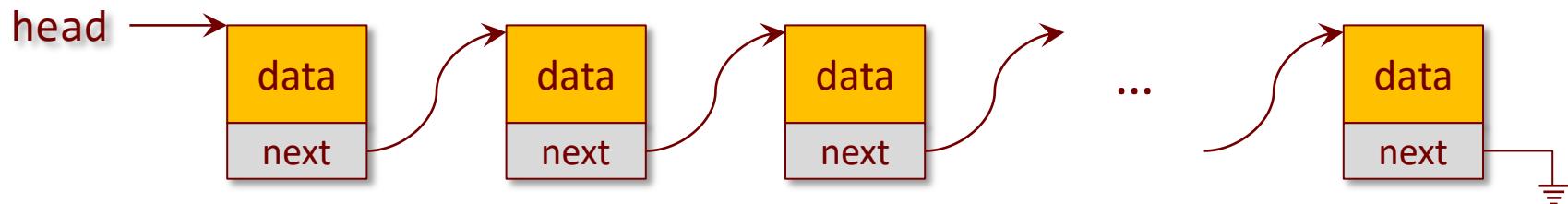
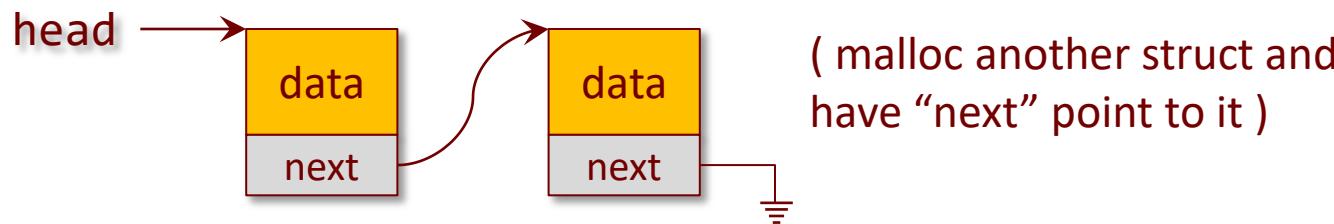
```
struct s1 {  
    .....  
    struct s2 *p;  
    .....  
};  
  
struct s2 {  
    .....  
    struct s1 *q;  
    .....  
};
```



Παράδειγμα – αναφορές / δυναμική λίστα (1/3)



Στον τελευταίο κόμβο πρέπει πάντα το “next” να είναι ίσο με NULL ώστε να βρίσκουμε που τελειώνει η λίστα.



Παράδειγμα – αναφορές / δυναμική λίστα (2/3)

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

struct listnode {          /* Define our simple struct */
    int value;              /* We store our data (a number) here */
    struct listnode *next;   /* Pointer to the next node in the list */
};

typedef struct listnode listnode_t;      /* for simplicity & clarity */

/* Calculate the sum of all numbers in the list */
void find_sum(listnode_t *node)
{
    int sum = 0;

    while (node != NULL) {      /* Traverse the list till its end */
        sum += node->value;     /* Value of current node */
        node = node->next;      /* Point to the next node */
    }
    printf("sum of numbers = %d\n", sum);
}
```

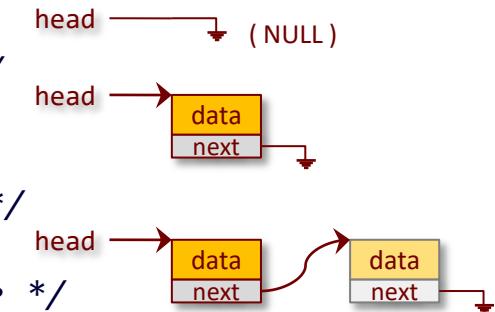
Παράδειγμα – αναφορές / δυναμική λίστα (3/3)

```
int main()
{
    listnode_t *node, *head = NULL; /* head: pointer to start of list (empty now) */
    int n;

    printf("How many numbers? ");
    scanf("%d", &n); /* This should be positive... */
    for (; n > 0; n--) {
        node = (listnode_t *) malloc(sizeof(listnode_t)); /* Malloc 1 node (struct) */
        if (node == NULL) return (1);

        scanf("%d", &node->value); /* Read & store number */

        if (head == NULL) { /* Empty list */
            node->next = NULL; /* No more nodes after this one */
            head = node; /* The only node in the list */
        }
        else { /* Non-empty list; insert new node */
            node->next = head; /* Inserted in the beginning */
            head = node; /* Head now points to the new node */
        }
    }
    find_sum(head); /* Call by passing the starting node of the list */
    return (0);
}
```



Η γλώσσα C

Ενώσεις (unions)



MYY502

Ενώσεις (Unions)

- ❖ Οι ενώσεις είναι μια ειδική περίπτωση «δομών» οι οποίες επιτρέπουν να κάνουμε οικονομία στη μνήμη.
 - Σε αντίθεση με τα structs, **δεν αποθηκεύονται όλα τα πεδία αλλά μόνο ένα από αυτά!!**
- ❖ Για παράδειγμα, ένα πρόγραμμα δέχεται ως είσοδο έναν int ή έναν double (ανάλογα με το τι επιλέγει ο χρήστης) – όχι και τα δύο μαζί. Πώς το αποθηκεύουμε αυτό;

```
int main() {                                     /* Απλή λύση: 2 μεταβλητές */
    int ival;
    float fval;
    char choice;
    scanf("%c", &choice);
    if (c == 'i') scanf("%d", &ival);
    if (c == 'f') scanf("%f", &fval);
    ...
}
```

- ❖ Μια ένωση αποτελείται από μια συλλογή πεδίων εκ των οποίων το πρόγραμμά μας μπορεί να επιλέξει ένα από όλα κάθε φορά.
- ❖ **Τα στοιχεία/πεδία μιας ένωσης μοιράζονται τον ίδιο χώρο μνήμης!!**
- ❖ Χειρισμός σαν να είναι structs.

```
typedef union trick {  
    int ival;  
    float fval;  
} trick;  
  
int main() {  
    trick value;      /* sizeof(value): 4 bytes, not 8 bytes! */  
    char choice;  
    scanf("%c", &choice);  
    if (c == 'i') scanf("%d", &value.ival);  
    if (c == 'f') scanf("%f", &value.fval);  
    ...  
}
```

Παράδειγμα – Ενώσεις

```
#define INT_TYPE 1
#define REAL_TYPE 2
struct item {
    int type;
    union {
        int ival;
        float fval;
    } value;
};

int main() {
    struct item x;

    x.type = INT_TYPE;
    x.value.ival = 4;
    printf("%d %f\n", x.value.ival, x.value.fval);
    x.type = REAL_TYPE;
    x.value.fval = 28000.5;
    printf("%d %f\n", x.value.ival, x.value.fval);
    return 0;
}
```

\$./a.out

4 0.000000

1188741376 28000.500000

Παράδειγμα – Ενώσεις

```
#define INT_TYPE 1
#define REAL_TYPE 2
struct item {
    int type;
    union {
        int ival;
        float fval;
    } value;
};

void print_item(struct item x) {
    if (x.type == INT_TYPE)
        printf("value = %d\n", x.value.ival);
    if (x.type == REAL_TYPE)
        printf("value = %f\n", x.value.fval);
}

int main() {
    struct item x = {INT_TYPE, { 4 }};
    print_item(x);
    return 0;
}
```

```
$ ./a.out
value = 4
```

Προγραμματισμός σε C

*Αρχεία κειμένου
(Text files)*



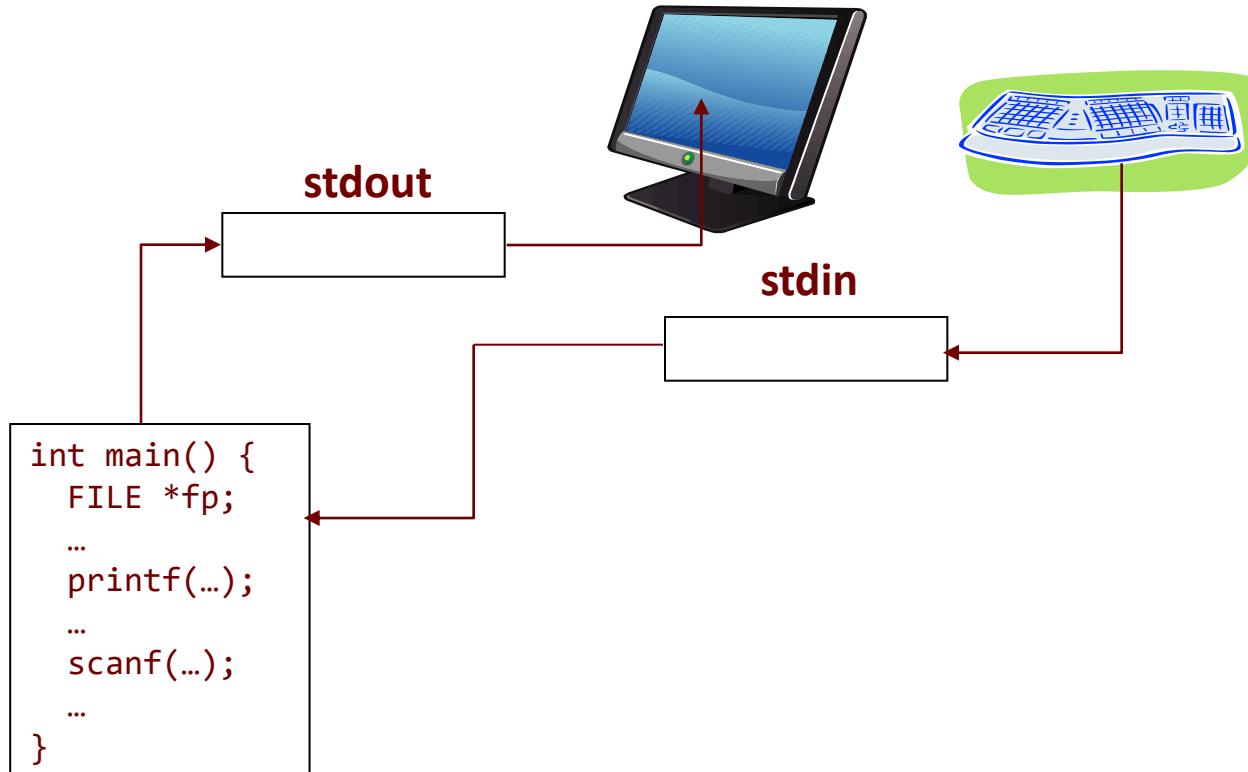
MYY502

Εισαγωγή

- ❖ Στη C έχουμε ειδικές συναρτήσεις για να επεξεργαζόμαστε αρχεία κειμένου που αποθηκεύονται στο δίσκο
 - Τα αρχεία είναι σημαντικά για μόνιμη αποθήκευση "κειμένου"
 - Αποθηκεύουν "απεριόριστη" πληροφορία.
 - Μπορείτε να τα θεωρήσετε ως ένα μεγάλο string που φυλάσσεται στον δίσκο και όχι στη μνήμη
 - Όμως ΔΕΝ είναι σαν τα strings διότι π.χ.
 - ✧ Το μέγεθός του μόνο αυξάνεται, δεν αφαιρείται τίποτε...
 - ✧ Δεν μπορείς να πας άμεσα σε όποιο στοιχείο (χαρακτήρα) θέλεις – συνήθως ξεκινάς από την αρχή και προχωράς ...
 - ✧ Επομένως δεν μπορείς να το χειριστείς με δείκτες.
- ❖ Προκειμένου να αποθηκευτεί / διαβαστεί πληροφορία σε αρχείο, απαιτείται η χρήση **ρευμάτων εισόδου/εξόδου (streams)**
 - ένα **ρεύμα εξόδου** μπορείτε να το φαντάζεστε σαν χώρο μνήμης στον οποίο το πρόγραμμα μόνο αποθηκεύει δεδομένα (χρησιμοποιώντας γνωστές συναρτήσεις – printf, fprintf, puts, fputs) τα οποία εν συνεχείᾳ τα παραλαμβάνει το λειτουργικό και τα στέλνει στο δίσκο.
 - ένα **ρεύμα εισόδου** μπορείτε να το φαντάζεστε σαν χώρο μνήμης στον οποίο το λειτουργικό αποθηκεύει δεδομένα από το δίσκο και εν συνεχείᾳ το πρόγραμμα μπορεί να παραλάβει αυτά τα δεδομένα για επεξεργασία χρησιμοποιώντας γνωστές συναρτήσεις (scanf, gets, fscanf, fgets...).

Ειδικά ρεύματα (streams)

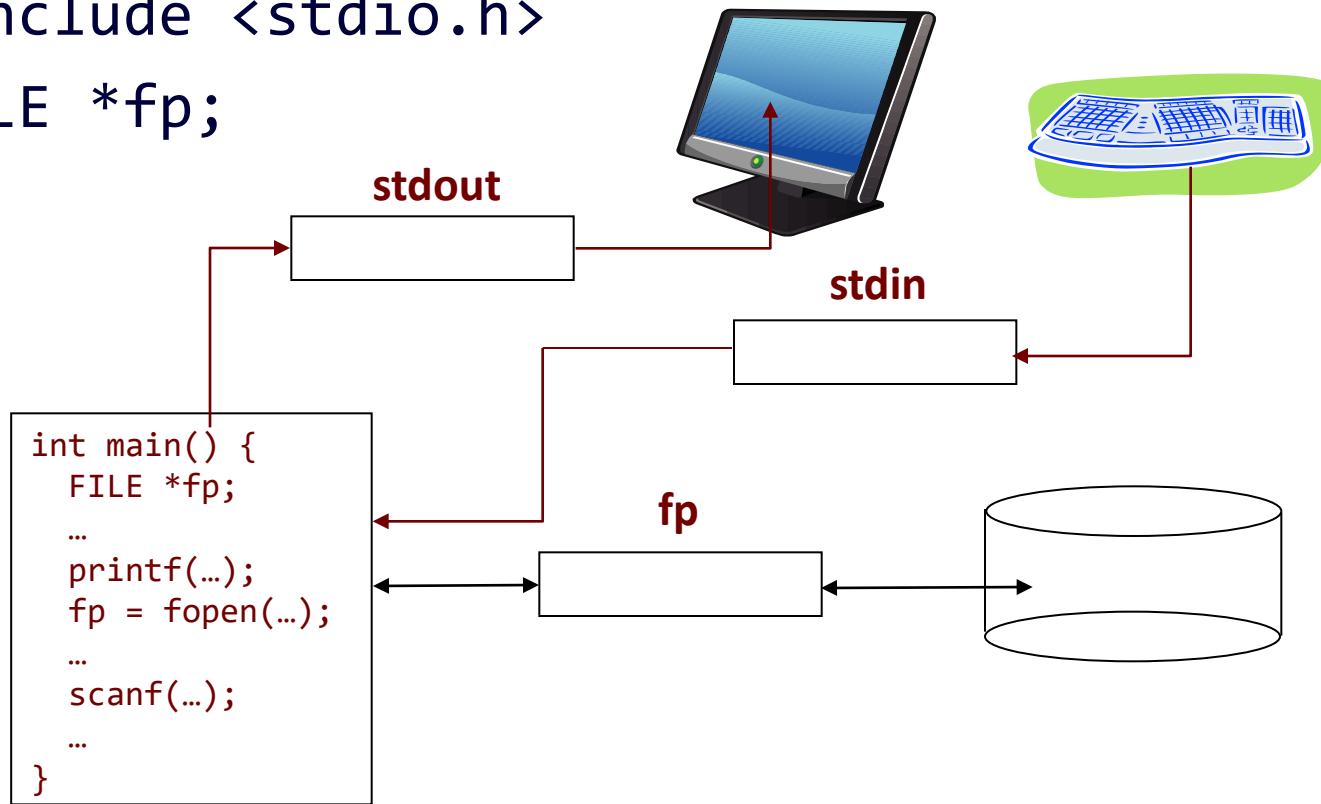
- ❖ **stdout** → γράφω σε ρεύμα που γίνεται flush στην οθόνη
- ❖ **stdin** → διαβάζω από ρεύμα στο οποίο γίνονται flush τα δεδομένα που περνάω από το πληκτρολόγιο



Ρεύματα που κατευθύνονται στο δίσκο

- ❖ Για να χρησιμοποιήσω από ένα πρόγραμμα κάποιο αρχείο χρειάζομαι ένα ρεύμα εισόδου/εξόδου το οποίο το δημιουργώ μέσω ενός δείκτη σε FILE.

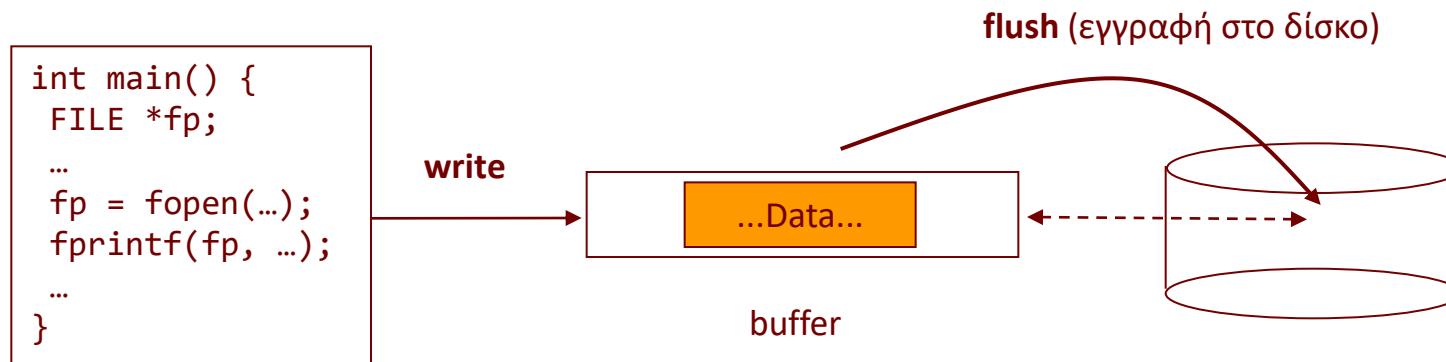
```
#include <stdio.h>
FILE *fp;
```



- ❖ FILE *fopen(char *name, char *mode);
 - name: όνομα αρχείου
 - mode:
 - ✧ "r" για ανάγνωση (αν δεν ξ, η fopen επιστρέφει NULL)
 - ✧ "w" για εγγραφή (αν ξ ήδη, τότε αδειάζουν τα περιεχόμενα)
 - ✧ "a" για επέκταση (αν ξ ήδη, τα περιεχόμενα διατηρούνται)
 - Άλλες επιλογές για mode
 - ✧ Η πρόσθεση του + μετά από έναν χαρακτήρα σημαίνει ότι επιτρέπεται και η αντίθετη χρήση ταυτόχρονα
 - ✧ "r+" για ανάγνωση / εγγραφή στην αρχή (αν δεν ξ, επιστρέφει NULL)
 - ✧ "w+" για ανάγνωση / εγγραφή με απόρριψη (αν ξ ήδη, άδειασμα)
 - ✧ "a+" για ανάγνωση / εγγραφή στο τέλος
- ❖ ΣΗΜΕΙΩΣΗ: τα ειδικά ρεύματα *stdout* και *stdin* είναι πάντα αρχικοποιημένα αυτόματα όταν ξεκινά ένα πρόγραμμα.

Αρχικοποίηση

- ❖ Η συνάρτηση fopen επιστρέφει έναν δείκτη (file pointer) σε ένα **ρεύμα** που συσχετίζεται με το αρχείο.
 - Άρα δεν μπορεί κανείς να προσβεί το αρχείο άμεσα μέσω του δείκτη (π.χ. δεν γίνεται, `*fp = 'a';`).
- ❖ Αν δεν υπάρχει αρχείο και ανοίγω με mode "w" ή "a", τότε δημιουργείται νέο αρχείο
- ❖ Αν κάτι πάει στραβά, η συνάρτηση επιστρέφει NULL



Βασικές Συναρτήσεις

- ❖ `int fscanf(FILE *fp, char *format, ...)`
 - Λειτουργεί όπως η scanf
 - Επιστρέφει αριθμό στοιχείων που διαβάστηκαν
 - Επιστρέφει EOF σε περίπτωση τέλους αρχείου ή λάθους
- ❖ `int fprintf(FILE *fp, char *format, ...)`
 - Λειτουργεί όπως η printf μόνο που γράφει στο fp
 - Επιστρέφει αριθμό στοιχείων που γράφτηκαν
 - Επιστρέφει < 0 σε περίπτωση λάθος
- ❖ Η scanf διαβάζει από τον ειδικό δείκτη αρχείου stdin, άρα:
`scanf("%d", &i);` ≡ `fscanf(stdin, "%d", &i);`
- ❖ Η printf γράφει στον ειδικό δείκτη αρχείου stdout, επομένως:
`printf("%d", i);` ≡ `fprintf(stdout, "%d", i);`



❖ `int fclose(FILE *fp);`

- Κλείνει το αρχείο (γίνεται flush o buffer)
 - Επιστρέφει 0 σε περίπτωση επιτυχίας
 - Επιστρέφει EOF σε περίπτωση λάθους
 - Το σύστημα εκτέλεσης κλείνει όλα τα ανοικτά αρχεία με τον τερματισμό του κυρίου προγράμματος. Αποτελεί καλή προγραμματιστική πρακτική, παρ' όλα αυτά, να κλείνουμε ρητά τα αρχεία που ανοίγουμε.
-
- ❖ Μπορεί να εκτελεστεί η εντολή `fclose(stdout)`, μετά όμως δεν μπορώ να γράψω – η `fprintf(stdout,...)` θα επιστρέφει **EOF**.

fgets

- ❖ `char *fgets(char *line, int maxline, FILE *fp);`
 - Διαβάζει το πολύ **maxline-1** χαρακτήρες
 - Αν βρει πρώτα αλλαγή γραμμής ('\n') διαβάζει μέχρι εκεί και η αλλαγή γραμμής ('\n') τοποθετείται στο line
 - Το line τερματίζει με το χαρακτήρα τερματισμού '\0'
 - Η συνάρτηση επιστρέφει σε περίπτωση λάθους NULL
- ❖ `char *gets(char *s);`
 - Η gets διαβάζει 1 γραμμή από το stdin ενώ αντικαθιστά τον τερματικό χαρακτήρα νέας γραμμής με '\0'.
 - Αν size of line < μέγεθος της γραμμής που διαβάζω από το stdin , τότε έχω undefined behavior
 - Συνιστάται η χρήση της fgets αντί της gets, όπως έχουμε πει
- ❖ `int fputs(char *s, FILE *fp);`
 - Γράφει το string s στο fp
 - Η συνάρτηση επιστρέφει τον αριθμό των χαρακτήρων που έγραψε και EOF σε περίπτωση λάθους
- ❖ `int puts(char *s);`
 - Γράφει το string s μαζί με τον χαρακτήρα αλλαγής γραμμής στο stdout

Λεπτομέρειες

- ❖ Ο δείκτης FILE *fp, δεν μας δίνει άμεση πρόσβαση στα περιεχόμενα του αρχείου (π.χ. ΔΕΝ ΜΠΟΡΟΥΜΕ ΝΑ ΚΑΝΟΥΜΕ ΑΡΙΘΜΗΤΙΚΗ ΜΕ AYTON). Άρα δεν μπορούμε να γράψουμε στο αρχείο με εντολές τύπου:
 - ~~— *fp = 'a'; /* Ο δείκτης δεν δείχνει στα περιεχόμενα */~~
 - ~~— strcpy(fp, "abc"); /* Το ίδιο */~~
- ❖ Τα περιεχόμενα τα διαβάζουμε / γράφουμε **ΜΟΝΟ** μέσω των fgets/fputs/fscanf/fprintf.
- ❖ Το αρχείο είναι σαν **TAINIA** ΑΠΟ ΚΑΣΕΤΑ. Όταν διαβάζουμε ή γράφουμε κάτι, **ΠΡΟΧΩΡΑΕΙ Η TAINIA ΑΠΟ ΜΟΝΗ ΤΗΣ.**
 - Επομένως, δεν προχωράμε εμείς τον δείκτη fp
 - Υπάρχει κλήση που μας επιστρέφει στην αρχή, ή σε κάποιο άλλο σημείο της "ταινίας".

Παράδειγμα 1 – fgets/fputs (διάβασμα γραμμή-γραμμή)

```
#include <stdio.h>

int main() {
    FILE *infile, *outfile;
    char buf[10];

    if ((infile = fopen("original.txt", "r")) == NULL)
        return 1;
    if ((outfile = fopen("copy.txt", "w")) == NULL)
        return 1;
    while (fgets(buf, 10, infile) != NULL) {
        fputs(buf, outfile);
    }
    fclose(infile);
    fclose(outfile);
    return 0;
}
```

Παράδειγμα 2 – fscanf/fprintf (διάβασμα λέξη-λέξη)

```
#include <stdio.h>

int main() {
    FILE *infile, *outfile;
    char buf[81];

    if ((infile = fopen("original.txt", "r")) == NULL)
        return 1;
    if ((outfile = fopen("copy.txt", "w")) == NULL)
        return 1;
    while (fscanf(infile, "%s", buf) != EOF) {
        fprintf(outfile, "%s", buf); /* Όλες οι λέξεις κολλητά */
        fprintf(stdout, "%s", buf); /* Εμφάνιση και στην οθόνη */
    }
    fclose(infile);
    fclose(outfile);
    return 0;
}
```

Παράδειγμα 3 – fflush/fclose

```
int main() {  
    FILE *fp;  
    fp = fopen("file", "w"); /* Should check for NULL */  
    fprintf(fp, "%s", "xxx");  
    while(1) ;  
    return 0;  
}
```

\$./a.out

^C

- ❖ Η εκτέλεση της εντολής "cat file" κατά την ώρα εκτέλεσης του προγράμματος δεν θα δείξει δεδομένα
- ❖ Το ίδιο θα συμβεί και μετά τον μη ομαλό τερματισμό του προγράμματος (με Control-C)
- ❖ Για την εγγραφή των δεδομένων πρέπει να προστεθεί η **fflush(fp);** πριν το while loop, η οποία **ολοκληρώνει** ότι **εγγραφές έχουν γίνει μέχρι εκείνη τη στιγμή στον δίσκο.**
- ❖ Η **fclose(fp);** κάνει ακριβώς το ίδιο (μόνο που επιπλέον κλείνει και το αρχείο).

Παράδειγμα 4 – Εύρεση μέγιστης λέξης

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

int main() {
    FILE *infile;
    char buf[101], maxWord[101];
    int maxLength = 0;

    if ((infile=fopen("testWords.ascii", "r")) == NULL) return 1;

    strcpy(maxWord, "");
    while (fscanf(infile, "%s", buf) != EOF) {
        if (strlen(buf)>maxLength) {
            strcpy(maxWord, buf);
            maxLength = strlen(buf);
        }
    }
    fprintf(stdout, "Max string is: %s with length %d\n",
            maxWord, maxLength);
    return 0;
}
```

Παράδειγμα 4 – Β' έκδοση, με χρήση stdio

```
/* Άμεση πληκτρολόγηση γραμμών κειμένου και τερματισμός με τον συνδυασμό Control-D (^D), που σηματοδοτεί το EOF */
#include <stdio.h>
#include <string.h>

int main() {
    char buf[101], maxWord[101];
    int maxLength = 0;

    strcpy(maxWord, "");
    while (fscanf(stdin, "%s", buf) != EOF) {
        if (strlen(buf)>maxLength) {
            strcpy(maxWord, buf);
            maxLength = strlen(buf);
        }
    }
    fprintf(stdout, "Max string is:%s with length %d\n",
            maxWord, maxLength);
    return 0;
}
```

Παράδειγμα 5 – fgets/sscanf (διάβασμα γραμμών + sscanf)

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

```
int main() {
    FILE *infile;
    int k;
    char s1[20], s2[20];
    float f;
    char buf[81];
```

```
if ((infile = fopen("testSscanf.txt", "r")) == NULL) return 1;
while (fgets(buf, 81, infile) != NULL) {
    puts(buf); /* show original line */
    sscanf(buf, "%d %s %s %f", &k, s1, s2, &f);
    printf("%d\t%s\t%s\t%f\n", k, s1, s2, f);
}
fclose(infile);
return 0;
}
```

```
$cat testSscanf.txt
2 minutes to 12.00
4 days to 1.0 breakdown
3 months to 5
$./a.out
2 minutes to 12.00

2      minutes to      12.000000
4 days to 1.0 breakdown

4      days      to      1.000000
3 months to 5

3      months   to      5.000000
```

Παράδειγμα 6 – μίνι grep

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

int main(int argc, char *argv[]) {
    FILE *infile;
    char buf[101];

    if ((infile = fopen("testWords.ascii", "r"))==NULL) {
        return 1;
    }

    if (argc != 2) {
        printf("Usage: %s searchString\n", argv[0]);
        return 1;
    }

    while (fgets(buf, 101, infile) != NULL) {
        if (strstr(buf, argv[1]) != NULL)
            printf("%s", buf); /* No "%s\n" here ?? */
    }
    fclose(infile);
    return 0;
}
```

Παράδειγμα 7 – Εκτύπωση λέξεων (fgets + strtok)

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

int main() {
    FILE *infile;
    int c;
    char buf[81];
    char *p, tokens[]={ "\n\t";

    if ((infile = fopen("testWords.ascii", "r"))==NULL) return 1;

    while (fgets(buf, 81, infile) != NULL) {
        c = buf[strlen(buf)-1];
        p = strtok(buf, tokens); /* εκτύπωση λέξεων γραμμής στην οθόνη */
        while (p) {
            printf("%s ", p);          /* ένα κενό μεταξύ των λέξεων */
            p = strtok(NULL, tokens);
        }
        if (c=='\n') printf("\n");
    }
    fclose(infile);
    return 0;
}
```

Παράδειγμα 8 – Μετρητής λέξης

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

void CheckDiomedes(char *X, int *counter) {
    char *c = strstr(X, "Diomhdh");

    while (c) {
        (*counter)++;
        c += strlen("Diomhdh");
        c = strstr(c, "Diomhdh");
    }
    printf("%s %d\n", X, *counter);
}

int main() {
    FILE *infile;
    char buf[81];
    int nOcc = 0;

    if ((infile = fopen("iliada.txt", "r"))==NULL)
        return 1;
    while (fscanf(infile, "%s", buf) != EOF)
        CheckDiomedes(buf, &nOcc);
    printf("Diomhdh: %d\n", nOcc);
    fclose(infile);
    return 0;
}
```

Παράδειγμα 9 – Ένωση αρχείων

```
/* Παράδειγμα εκτέλεσης: ./a.out f1.txt f2.txt bothfiles.txt */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *inp, *outp;
    char line[200];
    int i;

    if ((outp = fopen(argv[argc-1], "w")) == NULL) return 1;
    for (i = 1; i < argc - 1; i++) {
        if ((inp = fopen(argv[i], "r")) == NULL) continue;
        while (fgets(line, 200, inp) != NULL)
            fputs(line, outp);
        fclose(inp);
    }
    fclose(outp);
    return 0;
}
```

Θέση σε αρχείο

- ❖ Η δομή FILE διαχειρίζεται τη θέση του αρχείου στην οποία βρισκόμαστε αυτή τη στιγμή
 - Η αρίθμηση είναι από το 0 (αρχή του αρχείου, «πριν» τον πρώτο χαρακτήρα του αρχείου) μέχρι N, όπου N είναι το μέγεθος του αρχείου (θέση αμέσως μετά τον τελευταίο χαρακτήρα).
- ❖ Η θέση αυτή περιγράφεται από έναν ακέραιο long int.
 - Ουσιαστικά περιγράφει πόσους χαρακτήρες απέχουμε από την αρχή του αρχείου.
- ❖ Όπως γνωρίζουμε, αν καλέσουμε μια συνάρτηση εισόδου-εξόδου (π.χ. fscanf/fprintf), η τρέχουσα θέση αλλάζει αυτόματα μετά την ολοκλήρωσή της (προχωρά η "ταινία").
- ❖ Για να μάθουμε την τρέχουσα θέση:
 - long ftell(FILE *fp);
Επιστρέφει την τρέχουσα θέση στο αρχείο
- ❖ Μπορούμε να την τροποποιήσουμε με συγκεκριμένες εντολές

fseek / rewind

- ❖ `int fseek(FILE *fp, long offset, int pos);`
 - Η fseek ξεκινάει από την θέση **pos** και προσθέτει ή αφαιρεί **offset** θέσεις
 - Δυνατές τιμές για την pos
 - ✧ SEEK_CUR: τρέχουσα θέση στο αρχείο
 - ✧ SEEK_SET: αρχή αρχείου
 - ✧ SEEK_END: τέλος αρχείου
- ❖ `void rewind(FILE *fp);`
 - Τρέχουσα θέση = αρχή του αρχείου
 - `rewind(fp) ≡ fseek(fp, 0L, SEEK_SET);`
- ❖ Οι συναρτήσεις εγγραφής και ανάγνωσης (π.χ. fputs, fgets) αλλάζουν την τρέχουσα θέση.

Παραδείγματα fseek

- ❖ `fseek(fp, 0L, SEEK_SET)`
 - Στην αρχή του αρχείου
- ❖ `fseek(fp, 0L, SEEK_END)`
 - Στο τέλος του αρχείου (μετά τον τελευταίο χαρακτήρα)
- ❖ `fseek(fp, (long) -1, SEEK_END)`
 - Πριν τον τελευταίο χαρακτήρα. Η επόμενη ανάγνωση θα διαβάσει τον τελευταίο χαρακτήρα.
- ❖ `fseek(fp, ftell(fp), SEEK_SET)`
`fseek(fp, 0L, SEEK_CUR)`
 - Μείνε εδώ που είσαι (ισοδύναμα)
- ❖ Κατά το άνοιγμα του αρχείου με:
 - `FILE *fp = fopen("file", "r+");`
Η αρχική θέση θα είναι στην αρχή του αρχείου
 - `FILE *fp = fopen("file", "a+");`
Η αρχική θέση θα είναι στο τέλος του αρχείου

Παράδειγμα fseek

```
#include <stdio.h>
int main() {
    FILE *fp;
    char line[200];
    fp = fopen("file1.txt", "r+");
    fgets(line, 2, fp);
    fseek(fp, 4, SEEK_CUR);
    fputs("w", fp);
    fseek(fp, -1, SEEK_CUR);
    fgets(line, 200, fp);
    printf(stdout, "%s", line);
    return 0;
}
```

\$./a.out

wghijkl

\$cat file1.txt

abcdewghijkl

abcd~~e~~fghijkl

abcd~~e~~fghijkl

abcd~~e~~fghijkl

abcd~~e~~wghijkl

abcd~~e~~wghijkl

Παράδειγμα ftell

```
#include <stdio.h>
int main() {
    FILE *fp;
    char line[200];

    fp = fopen("file2.test", "r+");
    printf("%ld\n", ftell(fp));
    fseek(fp, ftell(fp)+4, SEEK_SET);
    fputs("w", fp);
    fseek(fp, ftell(fp)-1, SEEK_SET);
    fgets(line, 200, fp);
    fprintf(stdout, "%s", line);
    printf("%ld\n", ftell(fp));
    fseek(fp, 4, SEEK_SET);
    fgets(line, 200, fp);
    fprintf(stdout, "%s", line);
    fclose(fp);
    return 0;
}
```

\$cat file2.test

abcdefghijkl

\$./a.out

0

wabcdefghijkl

13

wabcdefghijkl

\$cat file2.test

abcdwabcdefghijkl

Προσθήκη δεδομένων ("a+")

```
#include <stdio.h>

int main() {
    FILE *fp;
    char buf[81] = "22 hours ago";

    fp = fopen("testMode.txt", "a+");
    if (fp == NULL) return 1;
    fputs(buf, fp);      /* γράφει στο τέλος του αρχείου */

    rewind(fp);
    while (fgets(buf, 81, fp) != NULL)
        fputs(buf, stdout); /* εκτύπωση στην οθόνη */
    fclose(fp);
    return 0;
}
```

```
$ cat testMode.txt
2 minutes to 12.00
4 days to 1.0 breakdown
3 months to 5
$ ./a.out
...
$ cat testMode.txt
2 minutes to 12.00
4 days to 1.0 breakdown
3 months to 5
22 hours ago
```

Προσθήκη δεδομένων ("r+")

```
#include <stdio.h>

int main() {
    FILE *fp;
    char buf[81] = "22 hours ago";

    fp = fopen("testMode.txt", "r+");
    if (fp == NULL) return 1;
    fputs(buf, fp);      /* γράφει στη τρέχουσα θέση, δηλαδή στην
                           αρχή του αρχείου */

    rewind(fp);
    while (fgets(buf, 81, fp) != NULL)
        fputs(buf, stdout);    /* εκτύπωση στην οθόνη */
    fclose(fp);
    return 0;
}
```

```
$ cat testMode.txt
2 minutes to 12.00
4 days to 1.0 breakdown
3 months to 5
$ ./a.out
...
$ cat testMode.txt
22 hours ago 12.00
4 days to 1.0 breakdown
3 months to 5
```

Προσθήκη δεδομένων ("w+")

```
#include <stdio.h>

int main() {
    FILE *fp;
    char buf[81] = "22 hours ago";

    fp = fopen("testMode.txt", "w+");
    if (fp == NULL) return 1;
    fputs(buf, fp); /* γράφει στην αρχή του αρχείου, όλα τα
                      προηγούμενα δεδομένα χάνονται */
    rewind(fp);
    while (fgets(buf, 81, fp) != NULL)
        fputs(buf, stdout); /* εκτύπωση στην οθόνη */
    fclose(fp);
    return 0;
}
```

```
$ cat testMode.txt
2 minutes to 12.00
4 days to 1.0 breakdown
3 months to 5
$ ./a.out
...
$ cat testMode.txt
22 hours ago
```

Προγραμματισμός σε C

*Πράξεις με bits
(bitwise operators)*



MYY502

Όλοι οι τελεστές για πράξεις με bits

Τελεστής	Περιγραφή
$x \& y$	AND bit-προς-bit
$x \mid y$	OR bit-προς-bit
$x \wedge y$	XOR bit-προς-bit
$\sim x$	Αντιστροφή των bits ενός αριθμού
$x \ll n$	Binary Left Shift Operator. Αριστερή μετατόπιση των bits ενός κατά n θέσεις.
$x \gg n$	Binary Right Shift Operator. Δεξιά μετατόπιση των bits ενός κατά n θέσεις.



Παράδειγμα

```
int main() {  
    unsigned int a = 60; /* 60 = 0011 1100 */  
    unsigned int b = 13; /* 13 = 0000 1101 */  
  
    int c = 0; c = a & b; /* 12 = 0000 1100 */  
    printf("Line 1 - Value of c is %d\n", c );  
  
    c = a | b; /* 61 = 0011 1101 */  
    printf("Line 2 - Value of c is %d\n", c );  
  
    c = a ^ b; /* 49 = 0011 0001 */  
    printf("Line 3 - Value of c is %d\n", c );  
  
    c = ~a; /*-61 = 1100 0011 */  
    printf("Line 4 - Value of c is %d\n", c );  
  
    c = a << 2; /* 240 = 1111 0000 */  
    printf("Line 5 - Value of c is %d\n", c );  
  
    c = a >> 2; /* 15 = 0000 1111 */  
    printf("Line 6 - Value of c is %d\n", c );  
    return 0;  
}
```

```
Line 1 - Value of c is 12  
Line 2 - Value of c is 61  
Line 3 - Value of c is 49  
Line 4 - Value of c is -61  
Line 5 - Value of c is 240  
Line 6 - Value of c is 15
```

Άλλα παραδείγματα

- ❖ Ο αριθμός **0...00100...0** (κ μηδενικά στα δεξιά του 1) – ποιος είναι;
 - Είναι ο 2^k . Πώς τον φτιάχνω;
 $x = (1 << k);$
- ❖ Ο αριθμός **1...11011...1** (κ μονάδες στα δεξιά του 0) – πώς τον φτιάχνω;
 $x = ~(1 << k); \quad /* \text{Αντιστρέφω τα bits του } 0...00100...0 */$
- ❖ Ο αριθμός **0...00111...1** (κ μονάδες) – πώς τον φτιάχνω;
 - Πρόκειται για τον αριθμό **0...00100...0** (κ μηδενικά στα δεξιά του 1) **ΜΕΙΟΝ 1**
 - $x = (1 << k) - 1; \quad /* \text{2-to-k minus 1} */$
- ❖ Διαιρεση δια / πολλαπλασιασμός επί 2^n :
 $x = x >> n; \quad y = y << n; \quad /* <<1 είναι πολ/μός επί 2 */$

- ❖ Κάνε το 1^o bit (από δεξιά) ίσο με 1:
 $x = x | 1; \quad /* \text{OR με το } 0...001 */$
- ❖ Κάνε το 1^o bit (από δεξιά) ίσο με 0:
 $x = x & (~1); \quad /* \text{AND με το } 1...110 */$
- ❖ Έλεγχος αν ο αριθμός είναι περιττός:
 $\text{if } (x \& 1) \quad /* \text{TRUE αν το 1}^{\circ}\text{ bit είναι 1 (περιττός)} */$



Κι άλλα παραδείγματα («μάσκες»)

- ❖ Οι μάσκες είναι αριθμοί που χρησιμοποιούνται για να εξάγουμε ή να θέσουμε σε κάποια τιμή τα bits ενός άλλου αριθμού.
- ❖ Ποιο είναι το λιγότερο σημαντικό bit ενός x ?
 $y = x \& 1;$ /* AND με το 000...01 (μάσκα) */

- ❖ Το 1^o byte ενός ακεραίου

$y = x \& 255;$ /* AND με το 000...01111111 */
 $y = x \& 0xFF;$ /* ισοδύναμο */

- ❖ Το 2^o byte ενός ακεραίου

$y = (x >> 8) \& 0xFF;$ /* ολίσθηση δεξιά 8 θέσεις */

- ❖ Θέσε το 6^o bit από δεξιά ίσο με 1:

$x = x | 32;$ /* OR με το $32 = 2^5 = 000...0100000$ */

- ❖ Μηδένισε το 6^o bit:

$x = x \& (\sim 32);$ /* AND με το 111...1011111 */

- ❖ Αντέστρεψε το 6^o bit:

$x = x ^ 32;$ /* XOR με το 000...0100000 */

Προγραμματισμός σε C

*Απαριθμήσεις
(enum)*



MYY502

Απαριθμήσεις (enums)

- ❖ Φανταστείτε ότι θέλουμε να ορίσουμε πολλές ακέραιες σταθερές με συνεχόμενες τιμές. Πώς το επιτυγχάνουμε αυτό; Με πολλά #define.
- ❖ Παράδειγμα: μία μεταβλητή φυλάει την ημέρα της εβδομάδας:

```
#define MON 1
#define TUE 2
#define WED 3
#define THU 4
#define FRI 5
#define SAT 6
#define SUN 7
```

```
int main() {
    int day = SUN;
    ...
}
```

Απαριθμήσεις (enum)

- ❖ Η C παρέχει τις απαριθμήσεις (enum) για ευκολότερο ορισμό. Πρόκειται για σύνολο συγκεκριμένων ακέραιων σταθερών.
- ❖ Βελτιώνει την αναγνωσιμότητα και δομή του προγράμματος μιας και φαίνεται σαν να είναι νέος τύπος δεδομένων (δεν είναι, είναι ακέραιοι)
- ❖ Για το προηγούμενο παράδειγμα:

```
/* Ορισμός του enum */
enum weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };

int main() {
    enum weekday day = WED;      /* Δήλωση μεταβλητής */
    ...
    if (day != SUN) day++;
    ...
}
```

Enum

- ❖ Αν δεν οριστεί τιμή για το πρώτο στοιχείο του συνόλου τότε θεωρείται ότι είναι το 0
- ❖ Μπορούμε να δώσουμε ότι τιμές θέλουμε στις σταθερές. Αν δεν δώσουμε, τότε συνεχίζουν από την τελευταία +1 κάθε φορά.
 - Π.χ. στο παρακάτω, τα third και fourth είναι το 5 και 6 αντίστοιχα:

```
enum ival { first=1, second = 4, third, fourth };
```
- ❖ Μπορεί να χρησιμοποιηθεί και με typedef:

```
enum weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };  
typedef enum weekday weekday_t;
```

```
int main() {  
    weekday_t day = WED;  
    ...  
}
```

Παραδείγματα με enum

❖ Ψευτο-boolean:

```
enum bool { FALSE, TRUE };  
typedef enum bool boolean;  
int main() {  
    boolean x;  
    x = TRUE;  
}
```

❖ Έχουν διαφορά τα δύο παρακάτω?

- A) enum weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };
- B) enum { MON = 1, TUE, WED, THU, FRI, SAT, SUN } weekday;

❖ ΝΑΙ, το A) ορίζει μία νέα απαρίθμηση και της δίνει το όνομα «weekday» ενώ το B) ορίζει μία **μεταβλητή** «weekday» τύπου απαρίθμησης (όπως και τα struct, έτσι και τα enum επιτρέπεται να μην έχουν όνομα).

Προγραμματισμός σε C

Συναρτήσεις με μεταβλητό πλήθος ορισμάτων
(Variadic functions)



MYY502

Συναρτήσεις με ακαθόριστο πλήθος παραμέτρων

- ❖ “Variadic”
- ❖ Π.χ. η `printf()`. Πόσα ορίσματα παίρνει;
- ❖ Στη C μπορούμε να ορίσουμε συναρτήσεις με άγνωστο πλήθος παραμέτρων/ορισμάτων.
 - Όμως πρέπει να υπάρχει τουλάχιστον 1 παράμετρος (δεν γίνεται να μην έχει καμία).
- ❖ Απαιτείται η χρήση του `#include <stdarg.h>`
 - Παλιά χρησιμοποιούσαμε το `<varargs.h>` αλλά όχι πλέον



Ορισμός συνάρτησης variadic

- ❖ Ορίζονται όπως όλες οι συναρτήσεις, όμως έχουμε δύο είδη παραμέτρων:
 - Πρώτα είναι οι **ΥΠΟΧΡΕΩΤΙΚΕΣ** παράμετροι (τουλάχιστον 1)
 - Στη συνέχεια τοποθετούνται οι **ΠΡΟΑΙΡΕΤΙΚΕΣ** (άγνωστο πλήθος), **απλά βάζοντας τρεις τελείες:**

```
int sum(int n, ...) { /* το n υποχρεωτικό */  
    <κώδικας>  
}
```

Κλήση συνάρτησης variadic

- ❖ Ακριβώς όπως οι κανονικές συναρτήσεις με όσες παραμέτρους θέλουμε (τουλάχιστον όσες και οι υποχρεωτικές):

```
int sum(int n, ...) { /* το n υποχρεωτικό */  
    <κώδικας>  
}  
  
int main() {  
    x = sum(2, 3, 5);      /* Άθροισε 2 αριθμούς */  
    y = sum(4, 1, 5, 7, 9); /* Άθροισε τέσσερις */  
    return 0;  
}
```

Μέσα στη συνάρτηση;

- ❖ Για να μπορέσουμε να βρούμε τις μη-υποχρεωτικές παραμέτρους, πρέπει να ορίσουμε μία μεταβλητή τύπου “va_list” και να την αρχικοποιήσουμε με την κλήση `va_start()`:

```
int sum(int n, ...) {      /* το n υποχρεωτικό */
    va_list args;
    int sum = 0;

    /* Αρχικοποίηση βάζοντας το όνομα της τελευταίας
       υποχρεωτικής παραμέτρου */
    va_start(args, n);

    <κώδικας>
}
```

Μη υποχρεωτικές παράμετροι

- ❖ Η προσπέλαση των παραμέτρων αυτών γίνεται η μία μετά την άλλη, συνήθως μέσα σε ένα loop. Για να πάρω την τιμή της επόμενης τέτοιας παραμέτρου, χρησιμοποιώ την `va_arg()` όπου είναι υποχρεωτικό να γνωρίζω και να αναγράψω τον ΤΥΠΟ της!
- ❖ Πριν την επιστροφή, πρέπει `va_end()`.

```
int sum(int n, ...) {      /* το n υποχρεωτικό */
    va_list args;
    int i, sum = 0, t;

    va_start(args, n);      /* Αρχικοποίηση */
    for (i = 0; i < n; i++) {
        t = va_arg(args, int);
        sum += t;
    }
    va_end(args);           /* Τέλος */
    return (sum);
}
```

- ❖ Πρέπει πάντα με κάποιο τρόπο να γνωρίζω το πλήθος των παραμέτρων αλλιώς μπορεί να «κρασάρει» το πρόγραμμα αν χρησιμοποιήσω την `va_arg()` και:
 - Οι παράμετροι έχουν τελειώσει ή
 - Η παρέμετρος δεν είναι του τύπου που βάλαμε στην `va_arg()`
- ❖ Για αυτό συνήθως
 - η πρώτη παράμετρος φροντίζει να καθορίζει πόσα είναι τα ορίσματα ή
 - βάζω στο τέλος μία παράμετρο-«σημάδι» ώστε αν φτάσω εκεί να τελειώσω.

Προγραμματισμός σε C

Δείκτες σε συναρτήσεις
(function pointers)



MYY502

Δείκτες σε συνάρτηση

- ❖ Η C επιτρέπει να έχουμε δείκτες που δείχνουν σε συναρτήσεις!
- ❖ Δηλώνονται ως:
 <τύπος επιστροφής> (*όνομα)(<παράμετροι>);
- ❖ Πού χρησιμεύουν; Παράδειγμα: Θέλω να κάνω μία συνάρτηση update()
η οποία αλλάζει τα στοιχεία ενός πίνακα.
 - Κάποιες φορές θέλω να τριπλασιάζονται οι τιμές των στοιχείων.
 - Κάποιες άλλες φορές θέλω να αντιστρέφονται οι τιμές των στοιχείων.
 - Κάποιες άλλες φορές θέλω να αλλάζουν πρόσημο
 - Κλπ κλπ.
- Πρέπει να κάνω διαφορετικές εκδόσεις της update() οι οποίες κάνουν ακριβώς τα ίδια πράγματα – διαφέρουν μόνο στην πράξη που κάνουν σε κάθε στοιχείο.

Πολλά αντίγραφα της update()

```
void update_triple(int n, float x[]) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = 3*x[i];  
}  
  
void update_inverse(int n, float x[]) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = 1/x[i];  
}  
  
void update_revsign(int n, float x[]) {  
    int i;  
    for (i = 0; i < n; i++)  
        x[i] = -x[i];  
}
```

- ❖ Όλα τα αντίγραφα είναι ολόιδια (εκτός από την πράξη στο κάθε στοιχείο του πίνακα).

Ευκολία: δείκτης σε συνάρτηση

- ❖ Κάνω μόνο 1 έκδοση της `update()` και βάζω ως επιπλέον όρισμα τη λειτουργία που πρέπει να γίνει στα στοιχεία. Οι λειτουργίες υλοποιούνται με ξεχωριστές συναρτήσεις:

```
float triple(float f) { return (3*f); }
float inverse(float f) { return (1/f); }
float revsign(float f) { return (-f); }

void update(int n, float x[], float (*operation)(float)) {
    int i;
    for (i = 0; i < n; i++)
        x[i] = (*operation)(x[i]); /* Η ισοδύναμα: operation(x[i]) */
}

int main()
{
    float array[5] = { 1.5, 2.5, 3.5, 4.5, 5.5 };
    float (*func)(float); /* Μεταβλητή func - δείκτης σε συνάρτηση */

    func = triple;          /* Δείχνει στη συνάρτηση triple */
    update(5, array, func);
    func = inverse;         /* Δείχνει στη συνάρτηση inverse */
    update(5, array, func);
    update(5, array, revsign);
    return 0;
}
```

Έτοιμη συνάρτηση στο stdlib.h

❖ Quicksort:

```
void qsort(void *start, int numelems, int size,  
          int (*compare)(void *, void *));
```

❖ Παράδειγμα κώδικα για ταξινόμηση ακεραίων κατά αύξουσα σειρά :

```
/* Πρέπει να επιστρέψει (όπως η strcmp()):  
   0 αν ίσα, < 0 αν a < b και > 0 αν a > b */  
int cmp(void *a, void *b) {  
    int x = *((int *) a),  
    int y = *((int *) b);  
    return ( x-y );  
}  
  
int main(int argc, char *argv[]) {  
    int array[100];  
    ...  
    qsort(array, 100, sizeof(int), cmp);  
    ...  
}
```

Προγραμματισμός σε C

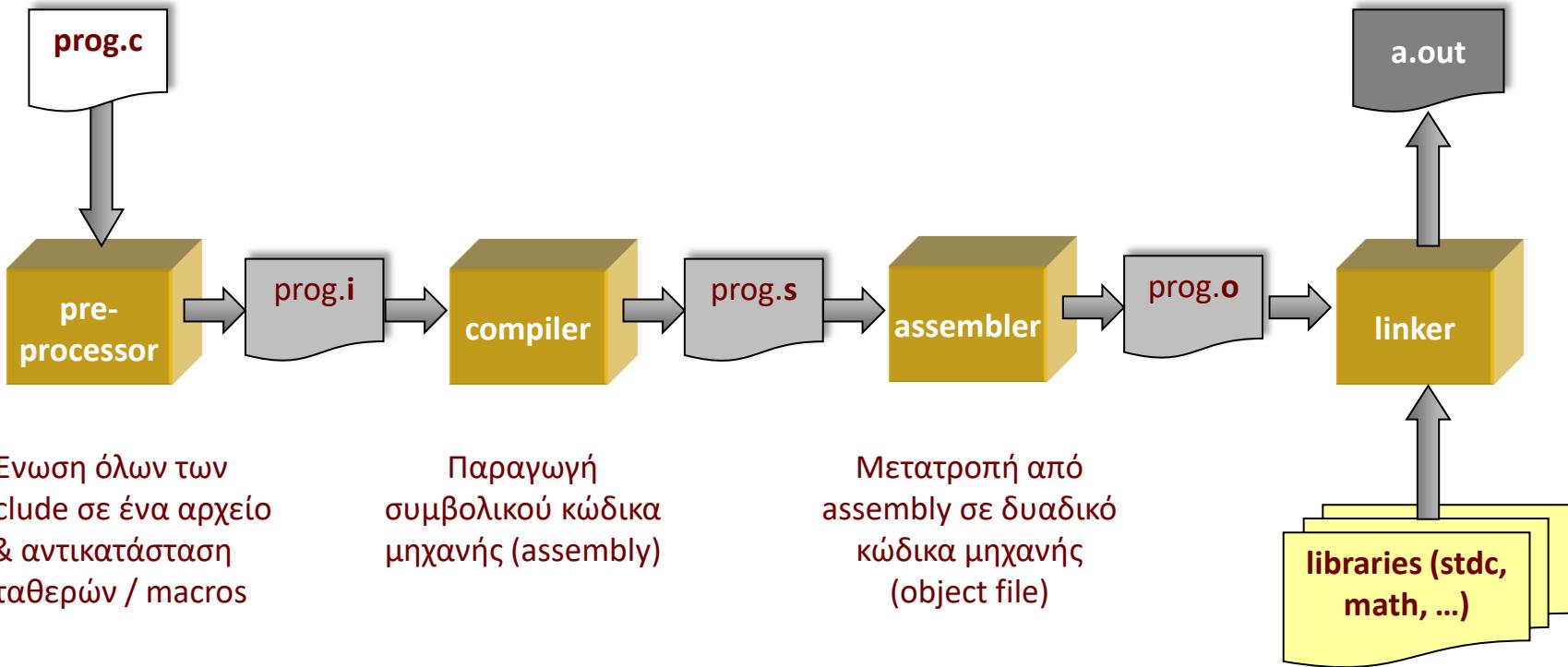
*Προχωρημένος προεπεξεργαστής
(C preprocessor)*



MYY502

Διαδικασία/Αλυσίδα μετάφρασης (π.χ. gcc)

- ❖ Τι γίνεται ακριβώς όταν μεταφράζεται ένα prog.c??



- Όλα τα ενδιάμεσα αρχεία μπορείτε να τα δείτε με: `gcc --save-temp prog.c`
- Για να τρέξει μέχρι τον preprocessor: `gcc -E prog.c`
- Για να τρέξει μέχρι τον compiler: `gcc -S prog.c`
- Για να τρέξει μέχρι τον assembler: `gcc -c prog.c`

1. «Πετάει» όλα τα σχόλια.
2. Ενώνει το πρόγραμμά μας μαζί με όλα τα αρχεία που κάνουμε `#include` σε ΕΝΑ αρχείο
3. Κάνει αντικατάσταση των σταθερών / macros (`#define`) που έχουμε στο πρόγραμμά μας
4. Το τελικό αποτέλεσμα τοποθετείται στο αρχείο `prog.i`
5. Το `prog.i` είναι αυτό που μεταφράζεται τελικά!

❖ Παρατηρήσεις:

- Οι σταθερές αντικαθίστανται ΠΡΙΝ το compile – άρα ο μεταφραστής δεν τις γνωρίζει!
- Ο preprocessor έχει μία μίνι-γλώσσα δική του, η οποία μπορεί να χρησιμοποιηθεί για διάφορα προχωρημένα τεχνάσματα.

Απλός preprocessor

- ❖ Ορισμός σταθερών – και όχι μόνο!!

```
#define N          0
#define MINUS1     4-5      /* ΔΕΝ κάνει υπολογισμούς */
#define KEYBOARD  stdin
#define then        /* τίποτε */
#define START      {
#define END        }
#define msg        "Please enter a positive number\n"
#define str         "hi!\n"
#define longloop   for (i = 0; i < n; i++)\
                  printf("%d ", i)

int main()
START
    int n = MINUS1, i;

    fscanf(KEYBOARD, "%d", &n);
    if (n < N)
        then {
            printf(str);
            printf(msg);
        END
    else START
        printf(str);
        longloop;
    }
    return 0;
END
```

Αποτέλεσμα preprocessor (.i)

```
int main()
{
    int n = 4-5, i;

    fscanf(stdin, "%d", &n);
    if (n < 0) {
        {
            printf("hi!\n");
            printf("Please enter a positive number\n");
        }
    else {
        printf("hi!\n");
        for (i = 0; i < n; i++) printf("%d ", i) ;
    }
    return 0;
}
```

Προχωρώντας ... macros με παραμέτρους

- ❖ Macro: παραμετροποιημένη έκφραση / αντικατάσταση

```
#define inch2cm(i) i*2.54
#define Square(x) x*x
int main() {
    int a, b;
    float l;
    a = Square(4);
    b = Square(a);
    l = inch2cm(2.0);
    ...
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {
    int a, b;
    float l;
    a = 4*4;
    b = a*a;
    l = 2.0*2.54;
    ...
}
```

- ❖ ΔΕΝ ΕΙΝΑΙ ΣΥΝΑΡΤΗΣΕΙΣ! Η αντικατάσταση γίνεται άμεσα (πριν τη μετάφραση).
- ❖ Το «όρισμα» της μακροεντολής αντικαθίσταται ως έχει
 - Αυτό δημιουργεί προβλήματα και θέλει προσοχή!

Παράμετροι και παρενθέσεις

- ❖ Τι θα γίνει παρακάτω;

```
#define Square(x) x*x

int main() {
    int a, b;
    a = Square(2+2);
    ...
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {
    int a, b;
    a = 2+2*2+2; /* 8 (όχι 16!) */
    ...
}
```

- ❖ **ΠΑΝΤΑ ΠΡΕΠΕΙ ΝΑ ΒΑΖΟΥΜΕ ΠΑΡΕΝΘΕΣΕΙΣ ΣΤΙΣ ΠΑΡΑΜΕΤΡΟΥΣ, όπου αυτές εμφανίζονται στον ορισμό του macro.**

Παράμετροι και παρενθέσεις

- ❖ Τι θα γίνει παρακάτω;

```
#define Square(x) (x)*(x)
#define Double(x) (x)+(x)
```

```
int main() {
    int a, b;
    a = Square(2+2);
    b = Square(-1);
    a = Double(a+b);
    b = 5*Double(a);
    ...
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {
    int a, b;
    a = (2+2)*(2+2); /* Σωστό */
    b = (-1)*(-1); /* Σωστό */
    a = (a+b)+(a+b) /* Σωστό */
    b = 5*(a)+(a) /* !!! */
    ...
}
```

- ❖ Όταν πρέπει, **ΒΑΖΟΥΜΕ ΠΑΡΕΝΘΕΣΕΙΣ ΚΑΙ ΣΕ ΟΛΗ ΤΗΝ ΕΚΦΡΑΣΗ ΤΟΥ MACRO**

```
#define Double(x) ((x)+(x))
```

Μακροεντολές, συνέχεια...

❖ Με περισσότερα ορίσματα:

```
#define max(a,b) ( ((a) > (b)) ? (a) : (b) )
```

```
int min(int a, int b) {  
    return ( (a < b) ? a : b );  
}
```

```
int main() {  
    int x = 4, y = 5;  
    x = max(x,y);  
    y = min(x,y);  
    return 0;  
}
```

Αποτέλεσμα preprocessor (.i)

```
int min(int a, int b) {  
    return ( (a < b) ? a : b );  
}
```

```
int main() {  
    int x = 4, y = 5;  
    x = ( ((x) > (y)) ? (x) : (y) );  
    y = min(x,y);  
    return 0;  
}
```

❖ Macro ή συνάρτηση?

- Εδώ η συνάρτηση είναι «χρονοβόρα» καθώς μεταφέρονται ορίσματα και γίνεται η κλήση της κατά τη διάρκεια εκτέλεσης του προγράμματος.
- Το συγκεκριμένο macro μεταφράζεται άμεσα και ο υπολογισμός του γίνεται χωρίς κλήση σε συνάρτηση.
- Το macro δεν γνωρίζει τύπους (και καλό και κακό...)

- ❖ Κατάργηση μίας σταθεράς / μακροεντολής από ένα σημείο και κάτω.

```
#define PI 3.14
```

```
int main() {  
    double x, y;  
    x = PI;  
#undef PI  
    y = PI;  
    ...  
}
```

Αποτέλεσμα preprocessor (.i)

```
int main() {  
    double x, y;  
    x = 3.14;  
  
    y = PI; /* Error: unknown  
              identifier */  
    ...  
}
```

Conditional compilation

- ❖ Πολλές φορές δεν γνωρίζουμε αν κάποιες σταθερές / συναρτήσεις υπάρχουν στο σύστημά μας. Π.χ. το NULL είναι ορισμένο κάπου? Το TRUE, το FALSE? Πρέπει να κάνουμε #include κάποιο header για να οριστεί?
- ❖ Αν πάμε και κάνουμε μόνοι μας π.χ. #define NULL 0, ενώ ήδη υπάρχει κάπου ορισμένο, τότε στην καλύτερη περίπτωση θα «γκρινιάξει» ο compiler για re-definition.
- ❖ Μπορούμε να κάνουμε κάτι για αυτό;
- ❖ Απάντηση: conditional compilation
 - Μπορούμε να ελέγχουμε αν κάτι έχει ήδη γίνει #define και να μην το ξανακάνουμε εμείς.
 - #ifdef / #ifndef / #else / #endif

Παραδείγματα conditional compilation

```
#ifdef FALSE
    /* Τίποτε */
#else
    #define FALSE 0
#endif

#ifndef TRUE
    #define TRUE 1
#endif

#ifdef __unix__                                /* Ανάλογα με το σύστημα */
    #include <unistd.h>
#else
    #ifdef _WIN32
        #include <windows.h>
    #endif
#endif
```

Γενικότερη σύνταξη

- ❖ Γενική σύνταξη

```
#if <condition>
    ...
#elif <condition>
    ...
#elif <condition>
    ...
...
#else
    ...
#endif
```

- ❖ Η συνθήκη είναι απλή έκφραση που εμπλέκει σταθερές συν το defined().
Παράδειγμα:

```
#if !defined(TRUE)      /* Ισοδύναμο με το #ifndef */
    #define TRUE 1
#endif
```

Γενικότερη σύνταξη

```
#if defined(NOTHING)
    #define DEBUG_LEVEL 0
#elif defined(SIMPLE) && !defined(ADVANCED)
    #define DEBUG_LEVEL 1
#elif defined(ADVANCED) || defined(Advanced)
    #define DEBUG_LEVEL 2
#else
    #define DEBUG_LEVEL 0
#endif

main() {
    ...
#if DEBUG_LEVEL==2
    printf("...
#endif
    ...
}
```

Τεχνική διατήρησης τμήματος κώδικα

- ❖ Θέλουμε το παρακάτω κομμάτι κώδικα να το «βάλουμε» σε σχόλια – δηλ. να μην το σβήσουμε (γιατί θα χρησιμοποιήσουμε στο μέλλον):

```
if (x > 0) { /* test for positive */  
    ...  
}  
else {           /* negative */  
    ...  
}
```

- ❖ Πώς; Αν απλά το βάλουμε σε σχόλια /* ... */, θα προκύψει λάθος από τη μετάφραση μιας και ήδη έχουμε σχόλια μέσα στο τμήμα αυτό του κώδικα.
- ❖ «Κλασική» λύση:

```
#if 0          /* Πάντα false => δεν μεταφράζεται! */  
if (x > 0) { /* test for positive */  
    ...  
}  
else {           /* negative */  
    ...  
}  
#endif
```

Προγραμματισμός συστημάτων UNIX/POSIX

*Ανακατευθύνσεις
(redirections)*



MYY502

- ❖ Κατά την εκτέλεση ενός προγράμματος, η είσοδος και η έξοδος ενός προγράμματος μπορούν να ανακατευθυνθούν από/σε κάποιο αρχείο της επιλογής μας. Π.χ. για χρήση του αρχείου “values” αντί του πληκτρολογίου, μπορούμε να εκτελέσουμε:

```
$ a.out < values
```

Όλες οι `scanf()/gets()` του προγράμματος αυτόματα διαβάζουν από το αρχείο αυτό και όχι από το πληκτρολόγιο.



- ❖ Για να τυπωθούν τα μηνύματα σε ένα αρχείο “res” αντί της οθόνης:

```
$ a.out > res
```

- ❖ Μπορούμε να δούμε τα αποτελέσματα απλά με ανάγνωση του αρχείου.
 - Όλα τα printf()/putchar()/puts() πάνε αυτόματα στο αρχείο αυτό αντί για την οθόνη

- ❖ Αν θέλουμε να δούμε τα λάθη του gcc όταν μεταφράζουμε ένα πρόγραμμα και είναι πολλά:

```
$ gcc test1.c > res
```

Δουλεύει;

Ανακατεύθυνση εξόδου

- ❖ Κάθε πρόγραμμα έχει 3 «αρχεία» ανοιχτά όταν εκτελείται:
 - Standard input (stdin)
 - Standard output (stdout)
 - Standard error (stderr)
- ❖ `fprintf(stderr, ...)`
- ❖ Χρησιμοποιείται από προγράμματα ώστε να μην μπλέκεται με τα τακτικά μηνύματα της εφαρμογής προς τον χρήστη.
- ❖ Τέλος, τα μηνύματα λάθους τα οποία τυπώνονται μεν στην οθόνη αλλά μέσω του stderr μπορούν επίσης να γραφτούν σε αρχείο με χρήση της εντολής:
`$a.out 2> err`
- ❖ Και το stdout και το stderr σε ένα αρχείο μαζί:
`$a.out &> res_and_err`

Πολλαπλές ανακατευθύνσεις

- ❖ Μπορεί κανείς να κάνει ταυτόχρονα πολλαπλές ανακατευθύνσεις:

```
a.out < values > results
```



Προγραμματισμός συστημάτων UNIX/POSIX

*Διαχείριση λαθών
(error handling)*



MYY502

errno

- ❖ Στα συστήματα POSIX, υπάρχει μία καθολική μεταβλητή η οποία προσδιορίζει το επακριβές σφάλμα που συνέβηκε στην τελευταία κλήση συστήματος στο πρόγραμμά μας.
- ❖ Πρέπει να κάνουμε #include <errno.h>
- ❖ Για παράδειγμα,

```
#include <errno.h> /* για το errno κλπ. */\n\nint main() {\n    FILE *fp;\n\n    fp = fopen("/tmp/tempfile", "r");\n    if (fp == NULL) {\n        fprintf(stderr, "Egine lathos: %d\n", errno);\n        exit (errno);\n    }\n    ...\n}
```

Συμβολικά

- ❖ Επειδή ο αριθμός δεν μας λέει και πολλά, μπορούμε να πάρουμε μία περιγραφή του λάθους με την perror():

```
#include <errno.h> /* για το errno κλπ. */

int main() {
    FILE *fp;

    fp = fopen("/tmp/tempfile", "r");
    if (fp == NULL) {
        perror("Egine lathos"); /* Εδώ θα προστεθεί το μήνυμα */
        exit (errno);
    }
    ...
}
```

- ❖ Υπάρχει πίνακας με τα μηνύματα αυτά, και χρησιμοποιείται το errno αυτόματα για να τυπωθεί το κατάλληλο.

Το μήνυμα του λάθους χωρίς να τυπωθεί.

- ❖ Μπορούμε απλά να πάρουμε το μήνυμα λάθους και να μην το τυπώσουμε (η perror() το τυπώνει) ώστε να το εμφανίσουμε με το δικό μας τρόπο
- ❖ Αυτό γίνεται με την strerror()

```
#include <errno.h> /* για το errno κλπ. */

int main() {
    char *msg;
    FILE *fp;

    fp = fopen("/tmp/tempfile", "r");
    if (fp == NULL) {
        msg = strerror(errno);
        fprintf(stderr, "Egine lathos (%d): [%s]", errno, msg);
        exit (errno);
    }
    ...
}
```

Προγραμματισμός συστημάτων UNIX/POSIX

*Δυαδικά αρχεία
(binary files)*



MYY502

Γενικά (Δυαδικά) αρχεία.

- ❖ Τα αρχεία που είδαμε μέχρι στιγμής (`fopen()`/`fclose()`) είναι απλά αρχεία κειμένου όπου μπορούμε να κάνουμε «φορμαρισμέμη» επικοινωνία (δηλ. διάβασμα/τύπωμα μέσω των `format` της `scanf/printf`).
- ❖ Βασικά δεν μπορούμε να κάνουμε και πολλά άλλα πράγματα...
- ❖ Δεν είναι όμως όλα τα αρχεία απλά αρχεία κειμένου. Π.χ. το `a.out` είναι αρχείο δυαδικό (*binary*).
- ❖ Τα αρχεία αυτά (αλλά γενικότερα οποιοδήποτε «αρχείο» – και το unix θεωρεί «αρχείο» τα πάντα, π.χ. τα αρχεία κειμένου, τα δυαδικά, τις συσκευές εισόδου/εξόδου, τα sockets, κλπ) τα χειριζόμαστε μέσω ειδικών συναρτήσεων «χαμηλότερου» επιπέδου.

Άδειες αρχείων

- ❖ ls -l
 - -rwxr-xr-x ... a.out ...
- ❖ Τα «rwxr-xr-x» περιγράφουν τι άδειες υπάρχουν για το αρχείο αυτό.
- ❖ Πρόκειται για 9 bits:

Bit	Σημασία
8	Ανάγνωση από τον κάτοχο
7	Εγγραφή από τον κάτοχο
6	Εκτέλεση από τον κάτοχο
5	Ανάγνωση από την ομάδα
4	Εγγραφή από την ομάδα
3	Εκτέλεση από την ομάδα
2	Ανάγνωση από τους υπόλοιπους
1	Εγγραφή από τους υπόλοιπους
0	Εκτέλεση από τους υπόλοιπους

Άδειες αρχείων

- ❖ Το «`rwxr-xr-x`» επομένως είναι ο δυαδικός αριθμός
 - 111101101
 - Στο οκταδικό, (0)755
- ❖ Δίνοντας κατάλληλες τιμές μπορούμε να καθορίσουμε ποιοι έχουν άδεια να κάνουν τι με κάποιο αρχείο μας.
- ❖ Με την εντολή `chmod` μπορούμε να αλλάξουμε τις άδειες αυτές (ο αριθμός πρέπει να είναι οκταδικός), π.χ.
 - `chmod 700 a.out`
 - `ls -l`
 - `-rwx----- ... a.out ...`

Περιγραφείς αρχείων

- ❖ Τα αρχεία που χειρίζεται ένα πρόγραμμα αριθμούνται με έναν απλό ακέραιο αριθμό (περιγραφέας – file descriptor).
- ❖ Κάθε εκτελούμενο πρόγραμμα έχει 3 αρχεία ανοιχτά αυτόματα, με περιγραφείς 0, 1, 2
 - 0: το standard input
 - 1: το standard output
 - 2: το standard error
- ❖ Μπορεί να ανοίξει κι άλλα (υπάρχοντα) με την *open()*
- ❖ Μπορεί να δημιουργήσει νέα αρχεία με την *creat()*
- ❖ Μπορεί να διαγράψει αρχεία με την *unlink()*
- ❖ Κλπ

open() – Άνοιγμα υπάρχοντος αρχείου

- ❖ **int open(char *path, int flags);**
- ❖ Η πρώτη παράμετρος είναι (ολόκληρο) το μονοπάτι που βρίσκεται το αρχείο.
- ❖ Η δεύτερη παράμετρος καθορίζει με τι τρόπο / λειτουργία θα προσπελαύνουμε το αρχείο (δηλ. ανάγνωση, εγγραφή ή και τα δύο).
 - O_RDONLY για μόνο ανάγνωση
 - O_WRONLY για μόνο εγγραφή
 - O_RDWR για ανάγνωση και εγγραφή
- ❖ Π.χ.
 - `int fd = open("/tmp/tempfile", O_RDWR);`
- ❖ Αν αποτύχει επιστρέφει αρνητικό αριθμό.

open() – Άνοιγμα υπάρχοντος αρχείου

- ❖ Παράδειγμα χρήσης:

```
#include <unistd.h> /* για την open() κλπ. */
#include <fcntl.h>   /* για τα flags O_XXX */

int main() {
    int fd;
    fd = open("/tmp/tempfile", O_RDWR);
    if (fd < 0)
        exit(1);
    ...
}
```

- ❖ Γιατί μπορεί να αποτύχει η open()?



open() – Δημιουργία νέου αρχείου

- ❖ Στην περίπτωση αυτή η open() πρέπει να καλείται με 3 ορίσματα:
 - `int open(char *path, int flags, mode_t permissions);`
- ❖ Ανάμεσα στα flags πρέπει να υπάρχει το `O_CREAT`.
- ❖ Όλα τα διάφορα flags που χρησιμοποιούνται πρέπει να γίνουν binary OR (|) μεταξύ τους. Π.χ.
 - `O_WRONLY | O_CREAT`
- ❖ Το `O_RDONLY | O_CREAT` δεν έχει πολύ νόημα αλλά όντως θα δημιουργηθεί κενό αρχείο στο οποίο δεν μπορούμε να γράψουμε τίποτε...
- ❖ Αν το νέο αρχείο που πάμε να δημιουργήσουμε υπάρχει ήδη, δεν θα γίνει τίποτε (απλά ανοίγει το αρχείο – αντίθετα αποτυγχάνει αν έχει δοθεί και `O_EXCL`). Αν όμως θέλουμε να σβηστεί το παλιό και να δημιουργήσουμε ένα νέο κενό αρχείο από την αρχή, πρέπει να δώσουμε και το flag `O_TRUNC`, π.χ
 - `O_WRONLY | O_CREAT | O_TRUNC`
- ❖ Η τρίτη παράμετρος καθορίζει τις **άδειες** που θα έχει το νέο αρχείο.
 - Αν και παλαιότερα ήταν ένας ακέραιος αριθμός (δινόταν ως οκταδικός συνήθως), πλέον είναι σύνολο από flags.

Ιστορική παρένθεση: creat() για δημιουργία νέου αρχείου

- ❖ Η περίπτωση όπου θέλουμε να δημιουργήσουμε νέο αρχείο και να σβηστεί το τυχόν παλιό αν υπάρχει είναι τόσο συνηθισμένη στην πράξη που υπάρχει ειδική συνάρτηση για αυτό, η creat().
- ❖ Είναι ΑΚΡΙΒΩΣ σαν την open() μόνο που παίρνει το 1^o και το 3^o όρισμα.
 - `int creat(char *path, mode_t permissions);`
- ❖ Είναι σαν να καλούμε την open() με δεύτερο όρισμα τον αριθμό:
`O_WRONLY | O_CREAT | O_TRUNC`
- ❖ Επομένως, την ξεχνάμε και θα χρησιμοποιούμε μόνο την open()
- ❖ **Ιστορικής σημασίας και μόνο.**

open() – Δημιουργία νέου αρχείου

- ❖ Η κλήση:
 - `int open(char *path, int flags, mode_t permissions);`
- ❖ Η τρίτη παράμετρος καθορίζει τις άδειες που θα έχει το νέο αρχείο.
 - Αν και παλαιότερα ήταν ένας ακέραιος αριθμός (δινόταν ως οκταδικός συνήθως), πλέον είναι σύνολο από flags.
 - Για τη χρήση του απαιτείται #include <sys/stat.h>

Bit	Σημασία	Τιμή για το mode_t
8	Ανάγνωση από τον κάτοχο	S_IRUSR
7	Εγγραφή από τον κάτοχο	S_IWUSR
6	Εκτέλεση από τον κάτοχο	S_IXUSR
5	Ανάγνωση από την ομάδα	S_IRGRP
4	Εγγραφή από την ομάδα	S_IWGRP
3	Εκτέλεση από την ομάδα	S_IXGRP
2	Ανάγνωση από τους υπόλοιπους	S_IROTH
1	Εγγραφή από τους υπόλοιπους	S_IWOTH
0	Εκτέλεση από τους υπόλοιπους	S_IXOTH

- ❖ Επομένως αντί για 0755, θα πρέπει να δώσουμε:

S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP |
S_IROTH | S_IXOTH

- ❖ Για την περίπτωση που ο χρήστης / η ομάδα / οι υπόλοιποι έχει όλες τις άδειες, μπορούμε αντίστοιχα να δώσουμε S_IRWXU, S_IRWXG ή/και S_IRWXO

- ❖ Επομένως, το παραπάνω συντομότερα είναι:

S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH



Διαγραφή αρχείου

- ❖ Για διαγραφή ενός αρχείου μπορεί να κληθεί η συνάρτηση:
`int unlink(char *path);`
- ❖ Αν το αρχείο είναι ανοικτό δε διαγράφεται άμεσα. Θα διαγραφεί μόλις κλείσει.

Κλείσιμο αρχείου

- ❖ Για να κλείσουμε ένα ανοιχτό αρχείο, πρέπει να κληθεί η συνάρτηση:

```
int close( int fd );
```

- ❖ Αν έχει προηγηθεί unlink(), το αρχείο θα διαγραφεί μετά την κλήση της close().

Γράψιμο στο αρχείο

- ❖ Για γράψιμο (αποθήκευση) στο αρχείο:

```
size_t write(int fd, void *buf, size_t nbytes);
```

- ❖ Παράδειγμα:

```
int arr[20];  
double x;
```

```
write(fd, &x, sizeof(double));  
write(fd, arr, 15*sizeof(int));
```

- ❖ Επιστρέφει το πλήθος των bytes που γράφτηκαν ή αρνητικό σε περίπτωση λάθους.

- ❖ **Πρέπει πάντα να γίνεται έλεγχος της τιμής επιστροφής!!!!**

- Δίσκος είναι, προβλήματα παρουσιάζονται...
- Μερικές φορές δεν γράφονται όλα τα bytes, πρέπει να ξαναγράψουμε τα υπόλοιπα.

Διάβασμα από αρχείο

- ❖ Για διάβασμα από το αρχείο:

```
size_t read(int fd, void *buf, size_t nbytes);
```

- ❖ Επιστρέφει το πλήθος (>0) των bytes που διαβάστηκαν, 0 για EOF, αρνητικό (<0) σε περίπτωση λάθους. Παράδειγμα:

```
int arr[20];
double x, sum = 0.0;

read(fd, arr, 15*sizeof(int));
while ( (n = read(fd, &x, sizeof(double))) > 0 ) {
    sum += x;
}
if (n < 0) { perror("Problem: "); exit(1); }
if (n == 0) /* EOF - End of File */
    close(fd);
```

Σχετική θέση αρχείου – lseek()

- ❖ Η θέση του αρχείου μετά το open() είναι στο αρχικό byte του (#0).
 - Αν έχουμε δώσει O_APPEND ανάμεσα στα flags της open(), θα είναι αμέσως μετά το τελευταίο byte του.
- ❖ Μπορούμε να αλλάξουμε τη θέση με τη συνάρτηση lseek():
`off_t lseek(int fd, off_t pos, int whence);`
(το off_t βασικά είναι ένας long int)
- ❖ Επιστρέφει τη νέα θέση.
- ❖ Στο pos δίνουμε πόσα bytes να προχωρήσει
- ❖ Το whence είναι ένα από:
 - SEEK_SET ώστε η απόλυτη νέα θέση να υπολογίζεται από την αρχή του αρχείου
 - SEEK_CUR ώστε η νέα θέση να υπολογίζεται από την τρέχουσα θέση του αρχείου
 - SEEK_END ώστε η απόλυτη νέα θέση να υπολογίζεται από το τέλος του αρχείου
- ❖ Αν η νέα θέση είναι ξεπερνά το μέγεθος του αρχείου, το αρχείο μεγαλώνει.

Παραδείγματα

```
off_t currentpos, filesize;

/* Πήγαινε στην αρχή */
lseek(fd, 0, SEEK_SET);
/* Πήγαινε στο τέλος (META το τελευταίο byte) */
lseek(fd, 0, SEEK_END);
/* Πήγαινε 4 bytes πριν */
lseek(fd, -sizeof(int), SEEK_CUR);

/* Εύρεση τρέχουσας θέσης & μεγέθους αρχείου σε bytes */
currentpos = lseek(fd, 0, SEEK_CUR);      /* Τρέχουσα θέση */
filesize = lseek(fd, 0, SEEK_END);          /* Στο τέλος */
lseek(fd, currentpos, SEEK_SET); /* Γύρνα εκεί που ήσουν */
```



Άδειασμα προσωρινής μνήμης

- ❖ Κατά το γράψιμο, δεν εγγράφονται όλα άμεσα στο δίσκο (στο `close()` εξασφαλίζεται ότι όλα θα γραφτούν).
 - Τοποθετούνται σε «προσωρινή μνήμη» και γράφονται από το λειτουργικό σύστημα κάποτε αργότερα
 - Για λόγους ταχύτητας
- ❖ Επειδή μπορεί να γίνει κάτι και να μην προλάβουν και γραφτούν τα δεδομένα μπορεί κάποιος να χρησιμοποιήσει τη συνάρτηση:
`void sync();`
 - Γράφει ότι δεν έχει γραφτεί ακόμα στο δίσκο, για όλα τα ανοιχτά αρχεία, όμως.
- ❖ Η συνάρτηση:
`int fsync(int fd);`
 - Γράφει ότι δεν έχει γραφτεί ακόμα στο δίσκο, για το συγκεκριμένο αρχείο.

Συγχρονισμένες εγγραφές

- ❖ Για να γίνεται πάντα **άμεση** εγγραφή στον δίσκο θα πρέπει κατά το `open()` να δώσουμε και το flag **O_SYNC**
 - Παράδειγμα:
`fd = open("/tmp/tempfile", O_RDWR | O_SYNC);`
- ❖ Ουσιαστικά θα καλείται αυτόματα η `fsync` μετά από κάθε εγγραφή με την `write`.
- ❖ Είναι πιο ασφαλές αλλά πολύ πιο αργό!

Δυαδικά αρχεία / αρχεία κειμένου

- ❖ Ποια θα είναι τα περιεχόμενα των αρχείων txtfile και binfile?

```
#include <stdio.h>

int main() {
    FILE *fp;
    int x = 1454654714;
    fp = fopen("txtfile", "w");
    fprintf(fp, "%d", x);
    fclose(fp);
    return 0;
}
```

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define Create (O_WRONLY|O_CREAT|O_TRUNC)
#define UserRW (S_IRUSR|S_IWUSR)

int main() {
    int fd;
    int x = 1454654714;
    fd = open("binfile", Create, UserRW);
    write(fd, &x, sizeof(int));
    close(fd);
    return 0;
}
```

Δυαδικά αρχεία / αρχεία κειμένου

- ❖ Το txtfile θα περιέχει:

```
$ cat txtfile
```

```
1454654714
```

- ❖ Γιατί;

- Διότι γράψαμε χαρακτήρες, ακριβώς όπως θα εμφανίζονταν και στην οθόνη.
- Το μέγεθος του αρχείου εξαρτάται από το πλήθος των ψηφίων του x!

```
#include <stdio.h>

int main() {
    FILE *fp;
    int x = 1454654714;
    fp = fopen("txtfile", "w");
    fprintf(fp, "%d", x);
    fclose(fp);
    return 0;
}
```



Δυαδικά αρχεία / αρχεία κειμένου

- ❖ To binfile θα περιέχει:

\$ cat binfile

VFILEDIRECTORY

- ❖ Γιατί;

- Διότι γράψαμε *bytes*
(πιθανώς μη εκτυπώσιμα)
ακριβώς όπως είναι
αποθηκευμένα στη
μνήμη.
- Το μέγεθος του αρχείου
θα είναι πάντα 4 bytes,
όσο πιάνει ένας ακέραιος
στη μνήμη.

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define Create (O_WRONLY|O_CREAT|O_TRUNC)
#define UserRW (S_IRUSR|S_IWUSR)

int main() {
    int fd;
    int x = 1454654714;
    fd = open("binfile", Create, UserRW);
    write(fd, &x, sizeof(int));
    close(fd);
    return 0;
}
```

Δυαδικά αρχεία / αρχεία κειμένου

- ❖ Στα αρχεία κειμένου ουσιαστικά σώζουμε ακριβώς ότι θα τυπώναμε και στην **οθόνη**, ενώ
 - στα δυαδικά αρχεία σώζουμε ακριβώς ότι υπάρχει στη **μνήμη** του υπολογιστή.
- ❖ Τα δυαδικά αρχεία τείνουν να είναι πιο μικρά και άρα πιο γρήγορα στην προσπέλασή τους
 - Χρήσιμο όταν έχουμε πραγματικά μεγάλα αρχεία
- ❖ Τα δυαδικά αρχεία δεν είναι εύκολα αναγνώσιμα και μεταφέρσιμα (portable)
 - Για μεταφέρσιμα αρχεία συνήθως χρησιμοποιούμε αρχεία κειμένου (π.χ. html).
 - Όμως, για μείωση χρόνου επικοινωνίας χρησιμοποιούμε συμπιεσμένα αρχεία, τα οποία είναι σχεδόν πάντα δυαδικά.
- ❖ Κάποια είδη αρχείων είναι μόνο δυαδικά (jpeg, mp3 κλπ).
- ❖ Κάποιες λειτουργίες του συστήματος (π.χ. sockets) είναι διαθέσιμες μόνο μέσω δυαδικών αρχείων.



Προγραμματισμός συστημάτων UNIX/POSIX

Διεργασίες
(processes)



MYY502

Δομή αρχείου προγράμματος

- ❖ **Πρόγραμμα** (program) ονομάζεται το εκτελέσιμο αρχείο που βρίσκεται αποθηκευμένο στο δίσκο (π.χ. το “a.out”, ή το “ls” ή το “gcc”)
 - Προγράμματα φτιάχνουμε εμείς (a.out) ή
 - Είναι έτοιμες εφαρμογές (/usr/bin/gcc, /bin/ls, ...)
- ❖ Τα προγράμματα είναι συνηθισμένα δυαδικά **αρχεία**. Απλά έχουν την ιδιότητα ότι μπορούν να **εκτελεστούν** (δηλαδή έχουν εντολές μηχανής).
- ❖ Τι περιέχει το αρχείο a.out? Ο μεταφραστής όταν δημιουργεί το πρόγραμμα, αποθηκεύει μέσα στο a.out διάφορα **τμήματα (segments)**:
 1. Τις **εντολές** του προγράμματος (σε γλώσσα μηχανής) – ονομάζεται **text** ή **code segment**
 2. Πληροφορίες για τις **αρχικοποιημένες καθολικές μεταβλητές και σταθερές** που έχει το πρόγραμμα (μεγέθη, αρχικές τιμές) – ονομάζεται **data segment**
 3. Πληροφορίες για τις **μη αρχικοποιημένες καθολικές μεταβλητές** που έχει το πρόγραμμα (μεγέθη) – ονομάζεται **bss segment**
- ❖ Μπορείτε να δείτε λεπτομέρειες των τμημάτων με την εντολή **size**, π.χ. εκτελέστε στο τερματικό:
 \$ size a.out

Παράδειγμα

```
int a = 5, arr1[50] = { 1, 2, 3 };
int b, arr2[50];
int main() {
    char *name = "Hi!";
    int c;
    c = a + b + strlen(name);
    return 0;
}
```

Αρχικοποιημένες καθολικές μεταβλητές και σταθερά strings βρίσκονται στο τμήμα data του a.out

Μη αρχικοποιημένες καθολικές μεταβλητές βρίσκονται στο τμήμα bss του a.out

Οι τοπικές μεταβλητές δεν βρίσκονται πουθενά! **Θα** τοποθετηθούν στη στοίβα όταν το πρόγραμμα εκτελείται ως διεργασία.

Οι εντολές βρίσκονται στο τμήμα text του a.out

Πρόγραμμα (program) & διεργασία (process)

- ❖ **Πρόγραμμα** (program) ονομάζεται το εκτελέσιμο αρχείο που βρίσκεται αποθηκευμένο στο δίσκο (π.χ. το “a.out”, ή το “ls” ή το “gcc”)
 - Προγράμματα φτιάχνουμε εμείς (a.out) ή
 - Είναι έτοιμες εφαρμογές (/usr/bin/gcc, /bin/ls, ...)
- ❖ Όταν ένα πρόγραμμα εκτελείται γίνεται **διεργασία** (process).
- ❖ Ανάλογα με το σύστημα, μπορεί να συνυπάρχουν και να εκτελούνται **πολλές διεργασίες σε κάθε χρονική στιγμή** (multitasking)
 - Π.χ. έχω ανοιχτό τον browser, τον editor και το τερματικό και την ίδια στιγμή ακούω κι ένα τραγούδι.
 - Μοιράζονται τον επεξεργαστή (ή τους επεξεργαστές), τη μνήμη, κλπ.
 - Αν υπάρχουν πολλοί επεξεργαστές (παράλληλοι υπολογιστές), μπορεί να εκτελούνται **ταυτόχρονα** αλλιώς μπορεί να εναλλάσσονται με πολύ γρήγορο ρυθμό και να εκτελούνται **εκ περιτροπής** δίνοντας την **ψευδαισθηση** της ταυτόχρονης εκτέλεσης (timesharing).

Διεργασίες

- ❖ Η διαδικασία κατά την οποία το πρόγραμμα μεταλλάσσεται σε διεργασία είναι ιδιαίτερα πολύπλοκη και την αναλαμβάνει το λειτουργικό σύστημα του υπολογιστή.
- ❖ «Χοντρικά» βήματα:
 1. Ανοίγεται το αρχείο του προγράμματος από τον δίσκο.
 2. Μεταφέρονται (αντιγράφονται) οι εντολές του (από το τμήμα text) σε κάποιο σημείο στην κύρια μνήμη
 3. Δεσμεύεται χώρος για την τοποθέτηση των καθολικών μεταβλητών (πληροφορίες από τα τμήματα data και bss)
 4. Αρχικοποιούνται οι καθολικές μεταβλητές (από το τμήμα data)
 5. Δεσμεύεται χώρος μνήμης για τη στοίβα του προγράμματος (stack)
 6. Προετοιμάζεται ο χώρος μνήμης από τον οποίο θα δίνονται bytes αν το πρόγραμμα καλεί την malloc (ο χώρος αυτός ονομάζεται σωρός – heap)
 7. Προετοιμάζονται πολύπλοκες δομές του λειτουργικού συστήματος για διαχείριση και δρομολόγηση της διεργασίας
 8. Εν τέλει, καλείται η main().



Μερικά για τις διεργασίες

- ❖ Μπορεί πολλές διεργασίες να προέρχονται από το **ίδιο** πρόγραμμα
 - Π.χ. έχω ανοιχτά πολλά τερματικά: ένα είναι το πρόγραμμα του τερματικού στον δίσκο, όμως εκτελούνται πολλές αυτόνομες και ξεχωριστές διεργασίες.
- ❖ Κάθε διεργασία παίρνει μία μοναδική **ταυτότητα (process id)** – είναι ένας ακέραιος αριθμός που της δίνει το λειτουργικό σύστημα
 - Η ταυτότητα είναι εν γένει τυχαία. Κάθε φορά που εκτελούμε το ίδιο πρόγραμμα, το σύστημα μπορεί να δίνει στην αντίστοιχη διεργασία διαφορετική ταυτότητα.
 - Διεργασίες που εκτελούνται ταυτόχρονα και οι οποίες προέρχονται από το ίδιο πρόγραμμα θεωρούνται εντελώς διαφορετικές, ανεξάρτητες και έχουν διαφορετικές ταυτότητες.
- ❖ Μπορώ να βρω ποιες διεργασίες εκτελούνται αυτή τη στιγμή, μαζί με τις ταυτότητές τους, εκτελώντας:
\$ ps

Πώς μπορώ να βρω την ταυτότητα της διεργασίας

- ❖ Η συνάρτηση **getpid()** επιστρέφει το process id της τρέχουσας διεργασίας, π.χ.

```
#include <stdio.h>
#include <unistd.h> /* For getpid() */

int main() {
    printf("Process id: %d\n", getpid());
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
Process id: 3757
$ ./a.out
Process id: 3760
```

- ❖ Μέσα από το πρόγραμμά μου μπορώ να εκτελέσω ένα οποιοδήποτε άλλο πρόγραμμα με την χρήση των κλήσεων **exec**.
 - Η διεργασία μου θα αντικατασταθεί και θα εκτελέσει το νέο πρόγραμμα.
 - Δηλαδή «εξαφανίζεται» ο κώδικας και τα δεδομένα της διεργασίας μου και στη θέση τους μπαίνουν ο κώδικας και τα δεδομένα του νέου προγράμματος που θέλω να εκτελέσω.
 - Το process id προφανώς δεν αλλάζει μιας και δεν φτιάχτηκε άλλη διεργασία. **Άλλαξε το πρόγραμμα αλλά όχι η διεργασία!!**
 - ✧ Επομένως, παρέμειναν πολλά πράγματα ΙΔΙΑ, όπως π.χ το user id, ο τρέχων φάκελος εργασίας, τα αρχεία που ήταν ανοιχτά κλπ
- ❖ Υπάρχουν αρκετές εκδόσεις της exec
 - exec(), execv(), execlpr(), execvpr(), execle(), execvne()
 - Κάνουν χοντρικά την ίδια δουλειά, αλλά με άλλα ορίσματα / επιλογές.

Η κλήση exec()

- ❖ Απαιτείται #include <unistd.h>
- ❖ Η execl() παίρνει ως ορίσματα **την πλήρη διαδρομή του προγράμματος** (αρχείου) που θέλω να εκτελέσω και στη συνέχεια οι επόμενες παράμετροι είναι όλα τα ορίσματα που πρέπει να περαστούν στην main() του προγράμματος που θα εκτελεστεί (με πρώτο το όνομα του προγράμματος, όχι την πλήρη διαδρομή του).
 - Η execl() προφανώς είναι variadic.
- ❖ Η τελευταία παράμετρος πρέπει να είναι NULL.
- ❖ Αν όλα πάνε καλά, η execl() δεν επιστρέφει ποτέ (έχει καταστραφεί η διεργασία που την κάλεσε...)
- ❖ Παράδειγμα: για να εκτελέσω “gcc –c myfile.c” θα πρέπει να καλέσω την execl() κάπως έτσι:

```
execl("/usr/bin/gcc", "gcc", "-c", "myfile.c", NULL);  
          ΠΛΗΡΗΣ ΔΙΑΔΡΟΜΗ   Όνομα   Όρισμα   Όρισμα   Τέλος
```

Παράδειγμα exec 1/3

```
#include <stdio.h>
#include <unistd.h> /* Needed for execl() */

int main() {
    printf("Before calling execl\n");
    /* Εκτέλεση "ls -l"; Δηλαδή "ls" με 1 όρισμα, το "-l" */
    execl("ls", "ls", "-l", NULL); /* NULL μετά το τελευταίο όρισμα */
    printf("After calling execl\n");
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

\$./a.out

Before calling execl

After calling execl

\$

- ❖ Τι έγινε???

To “ls” δεν είναι η ΠΛΗΡΗΣ διαδρομή της εντολής ls! Πρέπει να περάσουμε ολόκληρο το μονοπάτι όπου βρίσκεται αποθηκευμένο το πρόγραμμα ls.

Πώς το βρίσκουμε;

\$ which ls
/bin/ls
\$

Παράδειγμα exec 2/3

```
#include <stdio.h>
#include <unistd.h> /* Needed for execl() */

int main() {
    printf("Before calling execl\n");
    /* Εκτέλεση "ls -l"; Δηλαδή "ls" με 1 όρισμα, το "-l" */
    execl("/bin/ls", "ls", "-l", NULL);
    printf("After calling execl\n");
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
Before calling execl
total 7
-rwxr-xr-x 1 dimako staff 4688 May 14 22:23 a.out
-rw-r--r-- 1 dimako staff 167 May 14 22:23 test_exec.c
$
```

- ❖ Σωστή εκτέλεση! Το δεύτερο printf() δεν εκτελέστηκε ποτέ...



Παράδειγμα exec 3/3

- ❖ Το παρακάτω είναι ο κώδικας του προγράμματος a.out. Τι θα γίνει όταν το εκτελέσουμε;

```
#include <stdio.h>
#include <unistd.h> /* Needed for execl() */

int main() {
    printf("Before calling execl, pid = %d\n", getpid());
    execl("./a.out", "a.out", NULL);
    printf("After calling execl\n");
    return 0;
}
```

- ❖ Θα εκτελείται επ' άπειρο το a.out από την ίδια διεργασία και άρα:

```
$ ./a.out
Before calling execl 11454
...
...
```

Η κλήση execv()

- ❖ Πρόκειται για ίδια λογική με την `execl()`, μόνο που τα ορίσματα δεν δίνονται ξεχωριστά, αλλά μαζεμένα όλα μαζί σε έναν πίνακα:

```
#include <unistd.h>
int execv(char *path, char *argv[]);
```

- ❖ Το προηγούμενο παράδειγμα με χρήση της `execv()`:

```
#include <stdio.h>
#include <unistd.h> /* Needed for execl() */

int main() {
    /* Εκτέλεση "ls -l"; Πίνακας με 3 δείκτες σε συμβολοσειρές */
    char *arg[3] = {"ls", "-l", NULL };

    printf("Before calling execv\n");
    execv("/bin/ls", arg);
    return 0;
}
```

Προγραμματισμός συστημάτων UNIX/POSIX

*Δημιουργία νέων διεργασιών
(process creation - fork)*



MYY502

Δημιουργία νέας διεργασίας

- ❖ Μπορώ μέσα από το πρόγραμμά μου μπορώ να δημιουργήσω μία εντελώς νέα διεργασία;
 - Δυστυχώς οι συναρτήσεις της κατηγορίας `exec` ΔΕΝ δημιουργούν νέα διεργασία, παρά διατηρούν την τρέχουσα και απλώς τη βάζουν να εκτελέσει ένα άλλο πρόγραμμα...
- ❖ Υπάρχει ένας και μοναδικός τρόπος να γίνει αυτό: η (περίεργη) κλήση συστήματος `fork()`:
 - Η συνάρτηση αυτή δημιουργεί ένα **πανομοιότυπο αντίγραφο** της τρέχουσας διεργασίας και οι δύο, πλέον, διεργασίες εκτελούνται ταυτόχρονα και ανεξάρτητα!
 - Η διεργασία που κάλεσε την `fork()` ονομάζεται **διεργασία-γονέας (parent)** ενώ η νέα διεργασία ονομάζεται **θυγατρική ή διεργασία-παιδί (child process)**.

fork

- ❖ Όταν δημιουργηθεί η διεργασία-παιδί, δεν ξεκινάει να εκτελεί την *main()*. Αρχίζει να εκτελεί, **αμέσως μετά το σημείο που έγινε η κλήση στην *fork()*.**
 - Είναι λες και ο πατέρας και το παιδί να επιστρέφουν μαζί από την *fork*.
- ❖ Παράδειγμα:

```
#include <unistd.h> /* Needed for fork(), getpid() */  
  
int main() {  
    printf("parent (%d) about to call fork\n", getpid());  
    fork();  
    printf("Hi from process %d\n", getpid());  
    return 0;  
}
```

- ❖ Εκτέλεση στο τερματικό:
\$./a.out
parent (11533) about to call fork
Hi from process 11533
Hi from process 11534
\$

Εξήγηση

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

int main() {
    printf("parent (%d) about to call fork\n",
           getpid());
    fork();
    printf("Hi from process %d\n", getpid());
    return 0;
}
```

Η γονική διεργασία εκτελεί την printf

Η γονική διεργασία καλεί την fork

- 1) Δημιουργείται νέα διεργασία (θυγατρική). Πρόκειται για διεργασία-αντίγραφο του γονέα, η οποία εκτελεί το ίδιο πρόγραμμα.
- 2) Και η γονική και η θυγατρική διεργασία συνεχίζουν από την κλήση της fork και κάτω.

Αυτό το printf το εκτελούν και οι δύο διεργασίες (οι οποίες είναι διαφορετικές και άρα έχουν διαφορετικό process id)

```
$ ./a.out
parent (11533) about to call fork
Hi from process 11533
Hi from process 11534
$
```

Ερωτήματα & παρατηρήσεις

- ❖ Γιατί τυπώθηκε πρώτα το process id του γονέα και μετά του παιδιού; Προλαβαίνει πάντα και εκτελείται πρώτα ο γονέας;
 - Απάντηση: Είναι εντελώς τυχαίο! Αν ξαναεκτελέσουμε το a.out μπορεί να εκτυπωθεί πρώτα το μήνυμα του παιδιού...
- ❖ Δεν είναι μάλλον ανούσιο το γεγονός ότι γονέας και παιδί είναι πανομοιότυποι και κάνουν ακριβώς τα ίδια πράγματα;
 - Απάντηση: Προφανώς ναι! Μπορούμε όμως να τους βάλουμε να κάνουν διαφορετικά πράγματα.
- ❖ Πώς μπορεί το παιδί να γνωρίζει ότι είναι παιδί και να κάνει άλλα πράγματα από αυτά που κάνει ο γονέας;
 - Απάντηση: Πρέπει να κοιτάξουμε την τιμή που επιστρέφει η fork(). Αν είναι ίση με 0, τότε βρισκόμαστε στο παιδί, αλλιώς στον γονέα!
 - Στον γονέα, επιστρέφει το process id του παιδιού.

Διαφοροποίηση γονέα – παιδιού

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

int main() {
    int ret;
    printf("parent (%d) about to call fork\n", getpid());
    ret = fork(); /* Remember what fork() returned */
    if (ret == 0) /* 0 returned to child */
        printf("I am the child, with pid %d\n", getpid());
    else /* child's pid (> 0) returned to parent */
        printf("I am the parent, with pid %d; child pid = %d\n",
               getpid(), ret);
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
parent (11644) about to call fork
I am the parent, with pid 11644; child pid = 11645
I am the child, with pid 11645
$
```

getppid() – η ταυτότητα του γονέα

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

int main() {
    int ret;
    printf("parent (%d) about to call fork\n", getpid());
    ret = fork(); /* Remember what fork() returned */
    if (ret == 0) /* 0 returned to child */
        printf("I am the child, with pid %d; parent pid = %d\n",
               getpid(), getppid());
    else           /* child's pid (> 0) returned to parent */
        printf("I am the parent, with pid %d; child pid = %d\n",
               getpid(), ret);
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
parent (11640) about to call fork
I am the parent, with pid 11640; child pid = 11645
I am the child, with pid 11645; parent pid = 11640
$
```

Το παιδί είναι αντίγραφο του πατέρα

- ❖ Η θυγατρική διεργασία είναι πλήρες και πανομοιότυπο αντίγραφο της γονικής διεργασίας
- ❖ Τρέχει αντίγραφο του ίδιου προγράμματος με τον γονέα
- ❖ Ακόμα πιο σημαντικό: έχει πανομοιότυπα αντίγραφα όλων των μεταβλητών του γονέα, **με τις τιμές που είχαν στον πατέρα πριν τη fork()**.
- ❖ Γενικότερα, κληρονομεί τα πάντα από τον πατέρα. Ακόμα και όσα αρχεία είχε κάνει open() ο πατέρας, τα έχει και το παιδί και μάλιστα είναι ήδη ανοικτά!

Όλα ίδια – παράδειγμα

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

int a = 5;

int main() {
    int b = 10;
    printf("parent about to call fork\n");
    if (fork() == 0) /* 0 returned to child */
        printf("Child (pid %d): a = %d, b = %d\n", getpid(), a, b);
    else
        printf("Parent (pid %d): a = %d, b = %d\n", getpid(), a, b);
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
parent about to call fork
Child (pid 11704): a = 5, b = 10
Parent (pid 11703): a = 5, b = 10
$
```

Το παιδί είναι διαφορετική διεργασία

- ❖ Μπορεί να δημιουργείται ως πλήρες αντίγραφο του γονέα, όμως στη συνέχεια είναι αυτόνομο και
 - a) Μπορεί να εκτελεί άλλες εντολές του προγράμματος από ότι ο γονέας
 - b) Μπορεί να δίνει άλλες τιμές στις μεταβλητές, μιας και είναι δικές του, διαφορετικές από του γονέα.
- ❖ Ότι αλλαγές κάνει στις μεταβλητές του το παιδί, **δεν** επηρεάζουν τις αντίστοιχες μεταβλητές του πατέρα (και το αντίστροφο)

Διαφορετικές τιμές μεταβλητών – παράδειγμα

```
#include <stdio.h>
#include <unistd.h>
int a = 5;

int main() {
    int b = 10;
    if (fork() == 0) { /* 0 returned to child */
        printf("child: a = %d, b = %d\n", a, b);
        a++; b++;
        printf("child now: a = %d, b = %d\n", a, b);
    }
    else {
        printf("parent: a = %d, b = %d\n", a, b);
        a--; b--;
        printf("parent now: a = %d, b = %d\n", a, b);
    }
    return 0;
}
```

- ❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
parent: a = 5, b = 10
child: a = 5, b = 10
parent now: a = 4, b = 9
child now: a = 6, b = 11
$
```

Περιμένοντας το παιδί – waitpid()

- ❖ Όταν δημιουργείται μία νέα διεργασία, αυτή δρομολογείται και εκτελείται ανεξάρτητα από τον γονέα.
 - Μπορεί να τερματίσει ο γονέας και το παιδί να συνεχίσει να εκτελείται ή και το ανάποδο.
- ❖ Στις περισσότερες περιπτώσεις όμως, όταν φτιάξουμε μία νέα διεργασία, μας ενδιαφέρει να την περιμένουμε να ολοκληρώσει τη δουλειά της ή τέλος πάντων να γνωρίζουμε με κάποιο τρόπο ότι τερματίστηκε.
- ❖ Για αυτό το σκοπό χρησιμοποιείται η κλήση `waitpid()` η οποία μας επιτρέπει να σταματήσουμε και να περιμένουμε να τερματιστεί μία συγκεκριμένη διεργασία-παιδί

waitpid()

- ❖ Απαιτείται `#include <sys/wait.h>`
`pid_t waitpid(pid_t pid, int *exitstatus, int options);`
- ❖ Ο τύπος `pid_t` είναι κατά βάση ψευδώνυμο του `int`.
- ❖ Το πρώτο όρισμα (`pid`) είναι το process id της θυγατρικής διεργασίας που περιμένουμε να τελειώσει.
 - Αν περάσουμε **-1**, τότε περιμένουμε **μία οποιαδήποτε από τις θυγατρικές μας διεργασίες** να τελειώσει.
- ❖ Το τρίτο όρισμα δεν μας ενδιαφέρει – θα δίνουμε **πάντα 0**.
- ❖ Το δεύτερο όρισμα δείχνει σε έναν ακέραιο, στον οποίο θα αποθηκευτεί **η κατάσταση τερματισμού** (βασικά περιέχει την τιμή επιστροφής) της θυγατρικής διεργασίας.
 - Με αυτό τον τρόπο, το παιδί μπορεί επίσης να ενημερώσει τον γονέα για το τι έγινε.
 - Αν δεν μας ενδιαφέρει η τιμή, μπορούμε να περάσουμε `NULL`.

wait()

- ❖ Η κλήση

```
wait(&exitstatus);
```

- ❖ Είναι ισοδύναμη με την κλήση:

```
waitpid(-1, &exitstatus, 0);
```

- ❖ Δηλαδή, περιμένει τον τερματισμό οποιουδήποτε παιδιού.
- ❖ Την αγνοούμε επομένως...

Παράδειγμα αναμονής 1

```
#include <stdio.h>
#include <stdlib.h>      /* For exit() */
#include <unistd.h>      /* For fork(), getpid() */
#include <sys/wait.h>     /* For waitpid() */

int main() {
    int pid;

    pid = fork();
    if (pid == 0) { /* child */
        return 5;      /* child exits with 5 */
    }
    printf("parent (%d) waits for child (%d)... \n", getpid(), pid);
    waitpid(pid, NULL, 0);
    printf("child terminated! \n");
    return 0;
}
```

- ❖ Η θυγατρική διεργασία τερματίζει κανονικά με `return` από την `main()` της

Παράδειγμα αναμονής 2

```
#include <stdio.h>
#include <stdlib.h>      /* For exit() */
#include <unistd.h>      /* For fork(), getpid() */
#include <sys/wait.h>     /* For waitpid() */

void delay() {      /* Just delay */
    int i, sum=0;
    for (i = 0; i < 100000000; i++)
        sum += i;
    printf("child (%d) exits...\n", getpid());
    exit(5);      /* Child exits with 5 */
}

int main() {
    int pid;

    pid = fork();
    if (pid == 0)    /* child */
        delay();
    printf("parent (%d) waits for child (%d)... \n", getpid(), pid);
    waitpid(pid, NULL, 0);
    printf("child terminated!\n");
}
```

- ❖ Η θυγατρική διεργασία τερματίζει κανονικά με `exit()`

Παίρνοντας την τιμή τερματισμού

- ❖ Για να μπορέσω να βρω με τι τιμή τερμάτισε το παιδί μου (είτε με `exit` είτε με `return`), θα πρέπει να περάσω τη δεύτερη παράμετρο στην `waitpid()` με δείκτη σε κάποιον ακέραιο
- ❖ Στον ακέραιο αυτό, κατά την επιστροφή από την κλήση θα περιλαμβάνονται 2 ειδών πληροφορίες
 - α) Αν τερματίστηκε κανονικά (με `return` ή `exit`) η διεργασία-παιδί ή διακόπηκε απρόσμενα π.χ. λόγω κάποιου σφάλματος ή `ctrl-C` κλπ.
 - β) Η τιμή τερματισμού αν τερματίστηκε κανονικά
- ❖ Για να διαπιστώσω το α) πρέπει να χρησιμοποιήσω το macro `WIFEXITED()`.
- ❖ Αν αυτό μου δώσει `TRUE`, τότε το β) προέκυπτε από το macro `WEXITSTATUS()`.

Παράδειγμα αναμονής 3

```
#include <stdio.h>
#include <stdlib.h>      /* For exit() */
#include <unistd.h>      /* For fork(), getpid() */
#include <sys/wait.h>     /* For waitpid() */

void delay() {      /* Just delay */
    int i, sum=0;
    for (i = 0; i < 100000000; i++)
        sum += i;
    printf("child (%d) exits...\n", getpid());
    exit(5);      /* Child exits with 5 */
}

int main() {
    int pid, status;

    pid = fork();
    if (pid == 0)    /* child */
        delay();
    printf("parent (%d) waits for child (%d)... \n", getpid(), pid);
    waitpid(pid, &status, 0);
    if (WIFEXITED(status)) /* Terminated OK? */
        printf("child exited normally with value %d\n", WEXITSTATUS(status));
    else
        printf("child was terminated abnormally.\n");
    return 0;
}
```

Λεπτομέρειες – Παρατηρήσεις – waitid()

- ❖ To macro WEXITSTATUS() επιστρέφει **ΜΟΝΟ** τα 8 λιγότερο σημαντικά bits της τιμής τερματισμού του παιδιού
 - Επομένως, αν το παιδί θέλει να «πει» κάτι στον γονέα μέσω exit/waitpid, αυτό θα πρέπει να είναι ένας αριθμός μέχρι το 255.
- ❖ Η πιο σύγχρονη εκδοχή της waitpid() είναι η συνάρτηση waitid():
`waitid(P_PID, childpid, &info, WEXITED);`
Για αναμονή οποιουδήποτε παιδιού:
`waitid(P_ALL, 0, &info, WEXITED);`
όπου το info είναι τύπου `siginfo_t`.
 - Στο info.si_status μπορεί κανείς να βρει την τιμή τερματισμού του παιδιού, και μάλιστα **ΟΛΟΚΛΗΡΟ** τον ακέραιο (όχι μόνο τα 8 λιγότερο σημαντικά bits).

Από το τερματικό, για να βρούμε λεπτομέρειες και πληροφορίες για συναρτήσεις, χρησιμοποιούμε την εντολή `man`:

```
$ man waitid
```

- ❖ Μία πολύ συνηθισμένη περίπτωση στην πράξη είναι να θέλουμε να δημιουργήσουμε μία θυγατρική διεργασία η οποία θα εκτελέσει ένα άλλο πρόγραμμα και θα τερματίσει.
- ❖ Αυτό το σενάριο υλοποιείται με τη χρήση της `fork()` για να δημιουργηθεί η νέα διεργασία και στη συνέχεια με το παιδί να καλεί την `exec()` για να εκτελέσει άλλο πρόγραμμα
- ❖ Ο γονέας, ανάλογα με την περίπτωση, μπορεί να περιμένει ή όχι τον τερματισμό του παιδιού.



Παράδειγμα fork + exec

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void childfunc() {      /* Just delay */
    printf("child (%d) will fire up a firefox window!\n", getpid());
    execl("/usr/bin/firefox", "firefox", NULL);
    exit(5);
}

int main() {
    printf("about to fork...\n");
    if (fork() == 0) /* child */
        childfunc();
    printf("parent (%d) waits for child...\n", getpid());
    waitpid(-1, NULL, 0); /* or just wait(NULL) */
    return 0;
}
```

Η συνάρτηση `system()`

- ❖ Πρόκειται για μία συνάρτηση η οποία κάνει τη λειτουργία των `fork` + `exec` + `waitpid` με μία μόνο κλήση!
 - Δημιουργεί ένα παιδί, το οποίο εκτελεί το πρόγραμμα που θέλουμε (μέσω shell) και ο γονέας επιστρέφει μόλις τερματίσει το παιδί.
- ❖ Η συνάρτηση `system()`:
 - Η `system()` παίρνει ως όρισμα 1 συμβολοσειρά, η οποία περιέχει ΑΚΡΙΒΩΣ την εντολή που θα εκτελούσαμε στο τερματικό!
 - a) Μπλοκάρει (σταματά προσωρινά) την τρέχουσα διεργασία
 - b) δημιουργεί μία νέα διεργασία η οποία εκτελεί ένα κέλυφος (shell, π.χ. Bash)
 - c) το κέλυφος εκτελεί την εντολή που περάσαμε ως παράμετρο
 - d) τερματίζει η διεργασία του κελύφους και
 - e) συνεχίζει η αρχική μας διεργασία.
 - Απλός, γρήγορος αλλά και λειτουργικά περιοριστικός τρόπος.

Παράδειγμα με system()

❖ Παράδειγμα:

```
#include <stdio.h>
#include <stdlib.h> /* For system() */

int main() {
    printf("Before system()\n");
    system("ls -l"); /* Create shell to execute ls */
    printf("After system()\n");
    return 0;
}
```

❖ Εκτέλεση στο τερματικό:

```
$ ./a.out
Before system()
total 16
-rwxr-xr-x  1 dimako  staff  6748 May 14 14:49 a.out
-rw-r--r--  1 dimako  staff   236 May 14 14:49 test.c
After system()
```

Προγραμματισμός συστημάτων UNIX/POSIX

*Διαδιεργασιακή επικοινωνία: αγωγοί
(IPC – inter-process communication: pipes)*



MYY502

Επικοινωνία μεταξύ διεργασιών γονέα-παιδιού

- ❖ Κατά κάποιο τρόπο, θα δημιουργήσουμε ένα τύπο δυαδικού «αρχείου» στο οποίο θα έχουν πρόσβαση και οι δύο διεργασίες.
- ❖ Αρχικά, θα θεωρήσουμε ότι οι διεργασίες έχουν προέρθει από το ίδιο πρόγραμμα, με χρήση της `fork()` (μπορείτε να φανταστείτε γιατί;)
 - Διότι με την `fork` το παιδί κληρονομεί τα πάντα όπως είπαμε, ακόμα και τα αρχεία που έχει ανοίξει ο γονέας.
- ❖ Ο γονέας θα ανοίξει το αρχείο και το παιδί θα το έχει ήδη ανοιχτό όταν δημιουργηθεί με την `fork()`.
- ❖ Τα ειδικά αυτά αρχεία ονομάζονται **(ανώνυμοι) αγωγοί (unnamed pipes)**
 - Διότι κατά το άνοιγμα δεν δίνεται κανένα όνομα αρχείου.

Αγωγοί (pipe)

- ❖ Δημιουργία & άνοιγμα pipe:

```
#include <unistd.h>
int pipe(int pfd[2]);
```

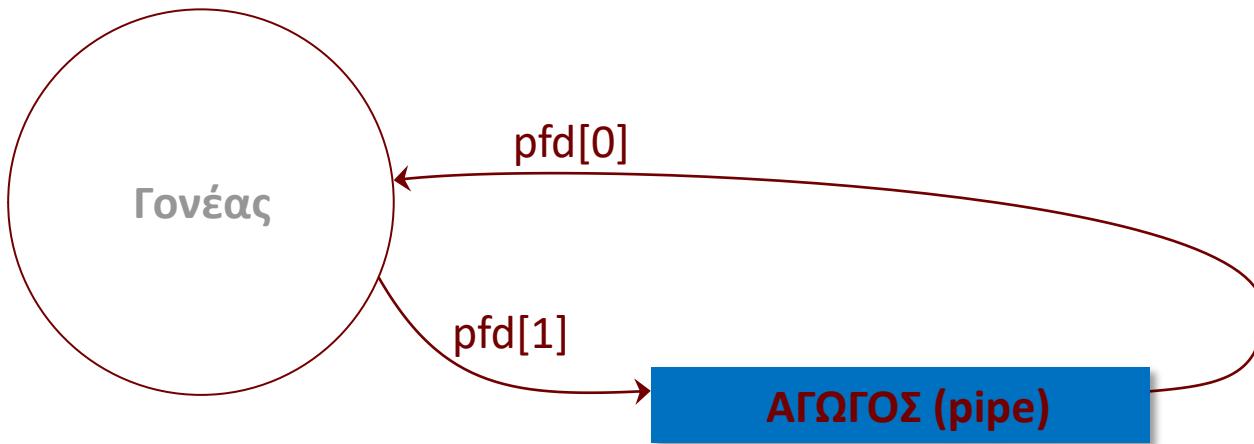
- ❖ Επιστρέφει 0 αν όλα OK – αλλιώς αρνητικό.
- ❖ Στο pfd[] ανοίγουν και επιστρέφονται 2 περιγραφείς αρχείων.
 - Με το pfd[1] μπορεί να γίνει ΜΟΝΟ γράψιμο (write) προς το pipe
 - Με το pfd[0] θα μπορεί να γίνει ΜΟΝΟ διάβασμα (read) από το pipe
 - Δηλαδή, ότι γράφεται στο pfd[1] διαβάζεται από το pfd[0].
- ❖ Εγγραφή και ανάγνωση γίνονται με τις γνωστές συναρτήσεις read() και write().
- ❖ Κλείσιμο γίνεται με την close().
- ❖ Δεν μπορείτε να κάνετε lseek().

Παράδειγμα με pipe

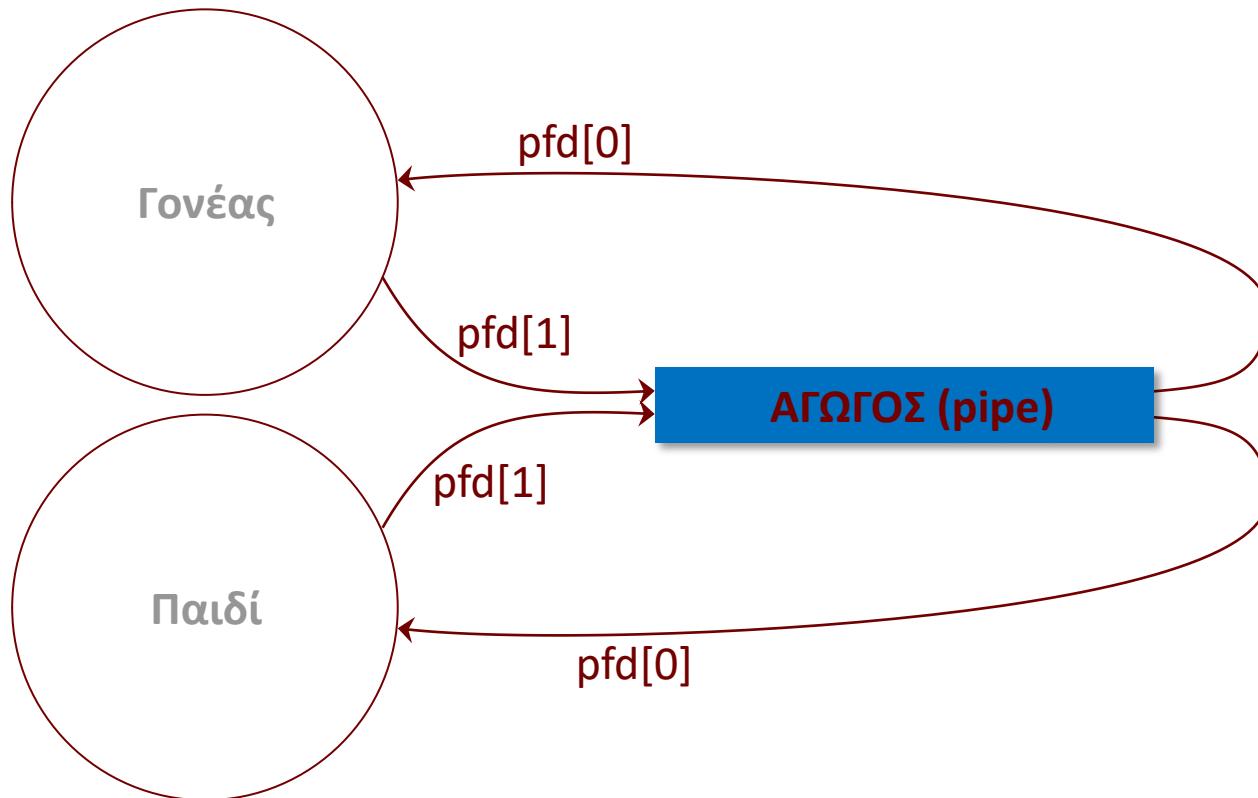
```
int main() {
    int n, pfd[2];
    char msg[50];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }
    if (fork() != 0) { /* Parent */
        close(pfd[0]);      /* No reading by the parent */
        printf("(%) - I am the parent; sending hello message.\n", getpid());
        write(pfd[1], "hello!!", 7);
    }
    else {                  /* Child */
        close(pfd[1]);      /* No writing by the child */
        printf("(%) - I am the child.\n", getpid());
        while ((n = read(pfd[0], msg, 2)) >= 0) {
            if (n == 0) break; /* EOF */
            printf("(%) - read %d bytes from the pipe [%.*s]\n", getpid(), n, n, msg);
        }
        if (n < 0) {
            perror("read() from child");
            exit(1);
        }
    }
    return 0;                /* All files close */
}
```

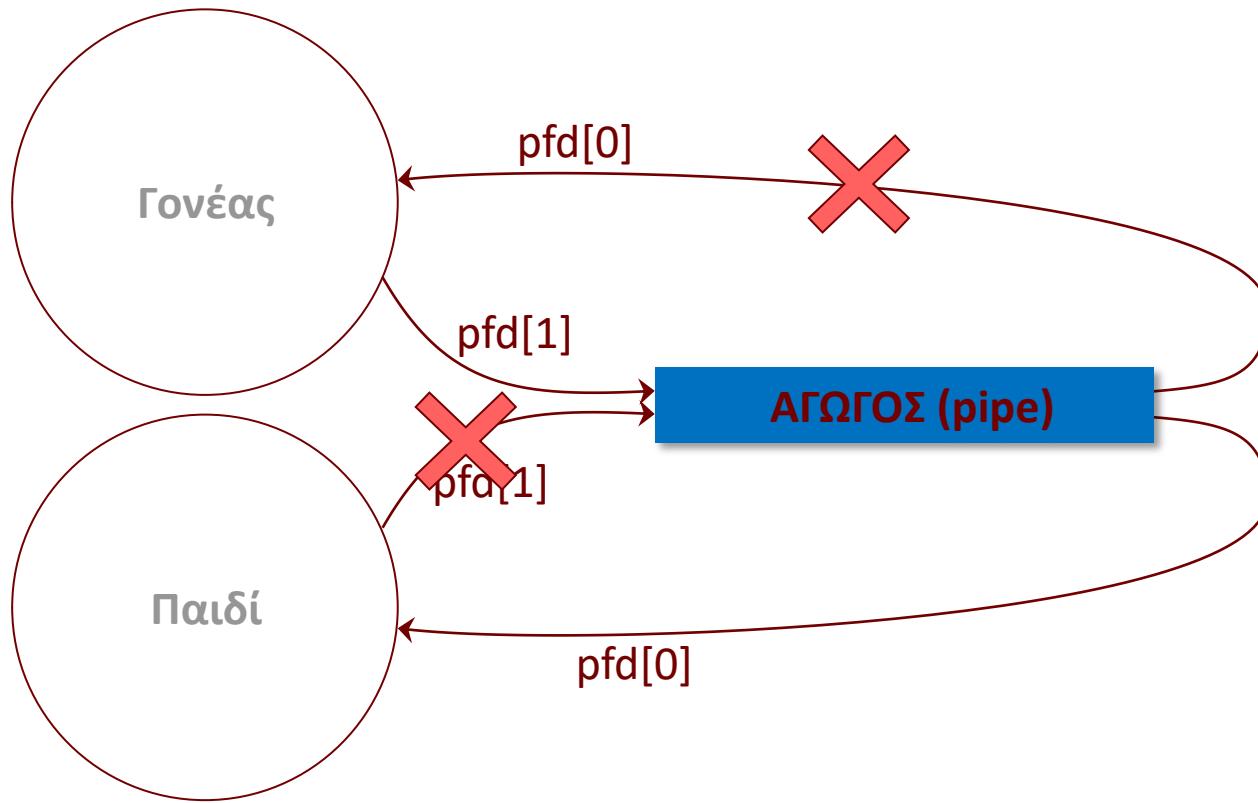
Παράδειγμα – δημιουργία pipe



Παράδειγμα – fork() και κληρονομιά pipe



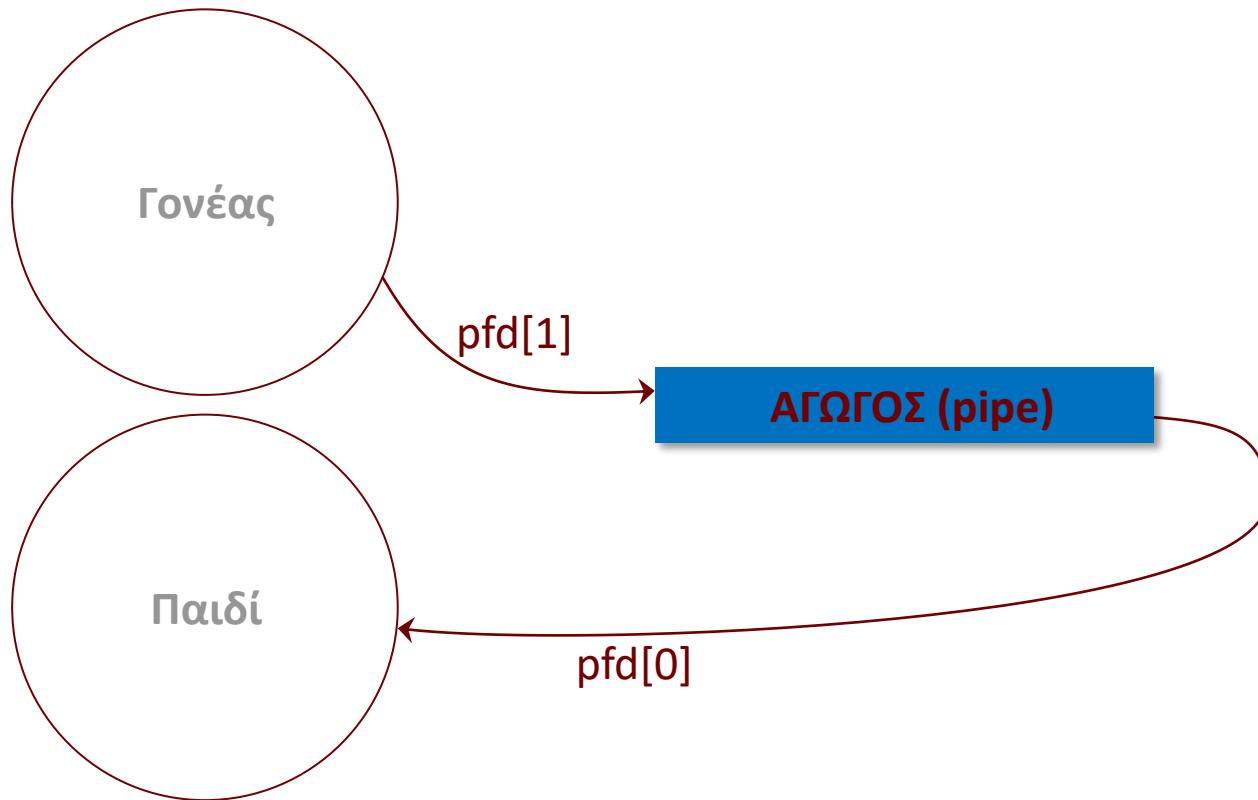
Παράδειγμα – κλείσιμο άχρηστων περιγραφέων



Πειράζει να μην κλείσω τους περιγραφείς;;;
Δεν είναι απαραίτητο να κλείσουν! Όμως, ανάλογα
με τι θέλουμε να κάνουνε, αν δεν κλείσουν μπορεί
να έχουμε προβλήματα (βλ. παρακάτω).



Παράδειγμα – τελική κατάσταση



Εκτέλεση παραδείγματος & λεπτομέρειες

❖ Εκτέλεση:

```
$ ./a.out
(23299) - I am the parent; sending hello message.
(23300) - I am the child.
(23300) - read 2 bytes from the pipe [he]
(23300) - read 2 bytes from the pipe [ll]
(23300) - read 2 bytes from the pipe [o!]
(23300) - read 1 bytes from the pipe [!]
```

❖ Μερικές λεπτομέρειες:

- Όταν ο εγγραφέας κάνει **close()**, στον αναγνώστη η **read()** θα επιστρέψει 0 (EOF-End of File), αφού όμως πρώτα διαβάσει όσα bytes είχαν ήδη γραφτεί.
 - ✧ Οποιοδήποτε δυαδικό αρχείο (άρα και ένας αγωγός) **παραμένει ανοικτό όσο υπάρχουν ανοικτοί περιγραφείς**
 - ✧ Για αυτό το παιδί κλείνει το **rfd[1]** όταν ξεκινάει – έτσι θα λάβει EOF μόλις κλείσει το **rfd[1]** και ο γονέας (εγγραφέας).
- Μπορεί να γράφει συνεχώς ο εγγραφέας ακόμα κι όταν δεν κάνει **read()** ο αναγνώστης;
 - ✧ Απάντηση: όχι – υπάρχει περιορισμένος χώρος (συνήθως 512 bytes). Αν γεμίσει, τότε ο εγγραφέας **μπλοκάρει** μέχρι ο αναγνώστης να αδειάσει χώρο με την **read()**.

❖ Τι γίνεται αν το παιδί κάνει εκεc??

- Τι γίνεται με τα ripes που έχει ανοίξει ο πατέρας;
- Η διεργασία-παιδί τα κληρονομεί OK. Όμως τα γνωρίζει το νέο πρόγραμμα που θα εκτελεστεί με την εκεc και αν ναι, πώς;

❖ Απάντηση:

- Όπως είχαμε πει, οι κλήσεις τύπου εκεc δεν δημιουργούν νέα διεργασία, αλλά διατηρούν την υπάρχουσα η οποία απλά εκτελεί άλλο πρόγραμμα
- Πολλά πράγματα όπως το process id, το user id, ο τρέχων κατάλογος εργασίας κλπ διατηρούνται.
- Όπως επίσης και οι ανοιχτοί περιγραφείς αρχείων (file descriptors)
- Οι οποίοι περιλαμβάνουν και τους περιγραφείς των αγωγών!
- Επομένως, το πρόγραμμα που θα τρέξει μέσω της εκεc θα έχει ανοιχτούς τους ίδιους περιγραφείς αρχείων όπως η διεργασία που το εκτελεί, συμπεριλαμβανομένων και των αγωγών.

❖ Υπάρχει όμως ένα πρόβλημα:

- OK, ο αγωγός θα είναι ανοικτός, αλλά πώς θα γνωρίζει το νέο πρόγραμμα ποιοι ακριβώς είναι οι περιγραφείς για τον αγωγό; (ώστε να μπορεί να γράψει / διαβάσει από αυτούς;)

❖ Δηλαδή, στο προηγούμενο παράδειγμα:

```
if (fork() != 0) { /* Parent */
    close(pfd[0]);      /* No reading by the parent */
    printf("(%) - I am the parent; sending hello message.\n", getpid());
    write(pfd[1], "hello!!", 7);
}
else {                  /* Child */
    close(pfd[1]);      /* No writing by the child */
    printf("(%) - I am the child. I will exec program 'pipe2'.\n", getpid());
    execl("./otherprog", "otherprog", NULL);
    printf("execl() failed!\n", getpid());
}
```

πώς θα γνωρίζει το otherprog ποιο είναι το pfd[0] ???

Περνώντας τον αγωγό μέσω exec

- ❖ Ο μόνος τρόπος να «περάσουμε» τον περιγραφέα στο πρόγραμμα που θα εκτελεστεί από την exec είναι:
μέσω ορισμάτων στην main του προγράμματος που θα εκτελεστεί
- ❖ Το πρόγραμμα θα πρέπει να έχει ορίσματα στην main() του, τα οποία θα του δώσουν τον ακέραιο αριθμό που αντιπροσωπεύει τον περιγραφέα του αγωγού.
 - Μην ξεχνάμε ότι ο αγωγός είναι ήδη ανοιχτός λόγω της κληρονομιάς της διεργασίας-παιδιού, απλά το νέο πρόγραμμα δεν ξέρει τον αύξων αριθμό του (περιγραφέα).

Παράδειγμα με exec & pipe, I

❖ Γονική διεργασία & παιδί που εκτελεί exec:

```
int main() {
    int pfd[2];
    char s[10];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }
    if (fork() != 0) { /* Parent */
        printf("(%d) - I am the parent; pfd[0]=%d,pdf[1]=%d.\n", getpid(),pfd[0],pfd[1]);
        close(pfd[0]);           /* No reading from the parent */
        printf("(%d) - I am the parent; sending hello message.\n", getpid());
        write(pfd[1], "hello!!", 7);
    }
    else {                  /* Child */
        close(pfd[1]);       /* No writing by the child */
        printf("(%d) - I am the child. I will exec otherprog.\n", getpid());
        sprintf(s, "%d", pfd[0]);           /* Write pfd[0] to a string */
        execl("./ otherprog", otherprog, s, NULL); /* Pass s as an argument */
        printf("exec() failed!\n");
    }
    return 0;
}
```

Παράδειγμα με exec & pipe, II

- ❖ To otherprog.c (μετάφραση με: gcc -o otherprog otherprog.c):

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

int main(int argc, char *argv[]) {
    int pfd, n;
    char msg[10];

    if (argc != 2)
        exit(1);
    pfd = atoi(argv[1]); /* Get the file descriptor */
    printf("(%) - I am %s. I will read from fd %d\n", getpid(), argv[0], pfd);

    while ((n = read(pfd, msg, 2)) >= 0) {
        if (n == 0) break; /* EOF */
        printf("(%) - read %d bytes from the pipe [%.*s]\n", argv[0], n, n, msg);
    }
    if (n < 0) {
        perror("read() from child");
        exit(1);
    }
    return 0;
}
```

Εκτέλεση παραδείγματος

```
$ ./a.out
(23859) - I am the parent; pfd[0]=3, pdf[1]=4.
(23859) - I am the parent; sending hello message.
(23860) - I am the child. I will exec otherprog.
(23860) - I am otherprog. I will read from fd 3
(pipe2) - read 2 bytes from the pipe [he]
(pipe2) - read 2 bytes from the pipe [ll]
(pipe2) - read 2 bytes from the pipe [o!]
(pipe2) - read 1 bytes from the pipe [!]
```



Σύνοψη επικοινωνίας με αγωγούς

1. Ο γονέας δημιουργεί και ανοίγει έναν αγωγό με την `pipe()`.
2. Επιστρέφονται 2 περιγραφείς, στον έναν γίνεται εγγραφή και από τον άλλον διάβασμα.
3. Δημιουργείται διεργασία-παιδί με τη `fork()`. Το παιδί κληρονομεί τους περιγραφείς.
4. Άμεσο κλείσιμο όσων περιγραφέων δεν χρειαζόμαστε (ώστε να ληφθούν EOF όταν πρέπει).
5. Διάβασμα και γράψιμο γίνεται σαν να πρόκειται για απλό αρχείο, μέσω `read()` και `write()`.
6. Κάθε διεργασία κλείνει με `close()`.
7. Αν το παιδί εκτελέσει μέσω εκτελεστή κάποιο άλλο πρόγραμμα, οι περιγραφείς μπορούν να περάσουν στο δεύτερο πρόγραμμα μέσω ορισμάτων στη `main()` του.

Θέματα με τους αγωγούς: αμφίδρομη επικοινωνία?

- ❖ Εκτός από τον γονέα, μπορεί να γράφει και το παιδί στον αγωγό για να τα διαβάζει ο γονέας;
- ❖ Απάντηση:
 - η σύντομη και ασφαλής απάντηση είναι: καλύτερα **'ΟΧΙ**,
 - παρότι υπό προϋποθέσεις μπορεί και να γίνεται.
- ❖ Οι αγωγοί θεωρούνται **μονό-κατευθυντήριες** επικοινωνίες.
- ❖ Πώς μπορεί λοιπόν να γράφει ο πατέρας προς το παιδί αλλά και το παιδί να γράφει προς τον πατέρα;
- ❖ Απάντηση:

Πρέπει να δημιουργηθούν **δύο ανεξάρτητοι αγωγοί**:

 - στον έναν θα γράφει ο πατέρας και θα διαβάζει το παιδί και
 - στον άλλον θα γράφει το παιδί και θα διαβάζει ο γονέας.

Θέματα με τους αγωγούς: αν δεν ελέγχουμε το exec?

❖ Όπως είπαμε πριν:

7. Αν το παιδί εκτελέσει μέσω exec κάποιο άλλο πρόγραμμα, οι περιγραφείς μπορούν να περάσουν στο δεύτερο πρόγραμμα μέσω ορισμάτων στη main() του.

Τι γίνεται αν το πρόγραμμα που εκτελείται μέσω της exec είναι μία έτοιμη εφαρμογή που δεν έχουμε πρόσβαση στον κώδικά της; Πώς μπορεί να ξέρει τον αγωγό μας ώστε να επικοινωνήσουμε μαζί της;

❖ Απάντηση:

Βασικά, δεν μπορούμε να κάνουμε τίποτε!

... και δεν γίνεται τίποτε;

- ❖ Ξέρουμε κάτι για κάθε εφαρμογή, ακόμα και για αυτές που δεν έχουμε γράψει εμείς;
- ❖ Μήπως όλες οι εφαρμογές γράφουν κάπου και διαβάζουν από κάπου;
- ❖ **Απάντηση:**
Πάντα έχουν ανοιχτά (υποχρεωτικά) δύο συγκεκριμένα αρχεία,
 - το standard input (με file descriptor 0, για διάβασμα από το πληκτρολόγιο) και
 - το standard output (με file descriptor 1, για γράψιμο στην οθόνη).
- ❖ Για λόγους φορητότητας (portability), αντί να χρησιμοποιούμε το 0 για το standard input και 1 για το standard output, πρέπει να χρησιμοποιούμε τα **STDIN_FILENO** και **STDOUT_FILENO** αντίστοιχα.

... και πώς μας βοηθάει αυτό να επικοινωνήσουμε;

- ❖ Από μόνο του δεν βοηθάει. Όμως υπάρχει η εξής ιδέα:
 - μπορούμε με κάποιον τρόπο να «συνδέσουμε» τους αγωγούς μας με την standard input ή/και την standard output της νέας διεργασίας;
 - Έτσι η διεργασία θα πιστεύει ότι διαβάζει π.χ. από το πληκτρολόγιο αλλά στην πράξη τα δεδομένα θα της τα δίνει ο γονέας μέσω ενός αγωγού.
- ❖ Απάντηση:

Αυτό όντως θα λειτουργήσει. Αρκεί βέβαια να μπορούμε να το κάνουμε!
- ❖ Απαιτείται να μπορούμε να κάνουμε «περίεργες» αλλαγές στους περιγραφείς...

Προγραμματισμός συστημάτων UNIX/POSIX

«Παιχνίδια» με περιγραφείς αρχείων &
αγωγούς



MYY502

«Παιχνίδια» με τους περιγραφείς αρχείων

- ❖ Υπάρχει η δυνατότητα να δημιουργήσουμε έναν **νέο (αντίγραφο) περιγραφέα** ο οποίος μπορεί να χρησιμοποιηθεί για προσπέλαση ενός **ήδη ανοιχτού αρχείου**.
- ❖ Δηλαδή, το *ίδιο αρχείο* μπορούμε να το προσπελάσουμε από δύο διαφορετικούς / ανεξάρτητους περιγραφείς.
- ❖ Μάλιστα μπορούμε να κλείσουμε τον πρώτο αλλά το αρχείο θα παραμείνει ανοιχτό μιας και είναι ανοιχτός ο δεύτερος.

- ❖ Δύο είναι οι κλήσεις που πετυχαίνουν τα παραπάνω:
 - `dup()`
 - `dup2()`

Η κλήση dup()

- ❖ Δημιουργία αντιγράφου περιγραφέα (υποτίθεται ότι το αρχείο με περιγραφέα fd είναι ήδη ανοιχτό):

```
#include <unistd.h>
int dup(int fd);
```

- ❖ Επιστρέφει έναν νέο περιγραφέα, ο οποίος μπορεί να χρησιμοποιηθεί να προσπελάσουμε το ίδιο αρχείο με τον fd.
- ❖ Για παράδειγμα, το παρακάτω γράφει στην οθόνη:

```
if ((new = dup(STDOUT_FILENO)) > 0) {
    write(new, "Hi!\n", 4);
    close(new);
}
```

Η κλήση dup2()

- ❖ Δημιουργία συγκεκριμένου αντιγράφου περιγραφέα (υποτίθεται ότι το αρχείο με περιγραφέα fd είναι ήδη ανοιχτό):

```
#include <unistd.h>
int dup2(int fd, int newfd);
```

- ❖ Ο νέος περιγραφέας θα είναι αυτός που δίνουμε (newfd)
- ❖ Επιστρέφει τον νέο περιγραφέα (δηλαδή το newfd) αν όλα πάνε καλά, αλλιώς έναν αρνητικό αριθμό.
- ❖ **Αν ο newfd χρησιμοποιείται ήδη, η dup2() πρώτα κλείνει το αρχείο του και στη συνέχεια κάνει τον newfd να προσπελαύνει το αρχείο του fd.**
- ❖ Για παράδειγμα, το παρακάτω γράφει στην οθόνη μέσω του 8:

```
if (dup2(STDOUT_FILENO, 8) > 0) {
    write(8, "Hi!\n", 4);
    close(8);
}
```

Παράδειγμα dup / dup2

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main() {
    int out, bak;

bak = dup(STDOUT_FILENO);      /* Backup standard output */
out = open("file.txt", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
if (bak < 0 || out < 0) exit(1);

printf("Hello world 1 \n");

dup2(out, STDOUT_FILENO);      /* out will be the new standard output */
close(out);
/* No longer needed */

printf("Hello world 2 \n");

dup2(bak, STDOUT_FILENO);      /* close & restore the original standard output */
close(bak);
/* No longer needed */

printf("Hello world 3 \n");
return 0;
}
```

Εκτέλεση:

```
$ ./a.out
Hello world 1
Hello world 3
$ cat file.txt
Hello world 2
```

Παράδειγμα με exec & pipe & dups, I

❖ Γονική διεργασία & παιδί που εκτελεί exec:

```
int main() {
    int pfd[2];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }
    if (fork() != 0) { /* Parent */
        printf("(%d) - I am the parent; pfd[0]=%d,pfd[1]=%d.\n", getpid(),pfd[0],pfd[1]);
        close(pfd[0]);      /* No reading from the parent */
        printf("(%d) - I am the parent; sending hello message.\n", getpid());
        write(pfd[1], "hello!!", 7);
    }
    else {                  /* Child */
        close(pfd[1]);      /* No writing by the child */
        if (dup2(pfd[0], STDIN_FILENO) >= 0) { /* Duplicate pipe to 0 (std input!) */
            execl("./pipe4", "pipe4", NULL);
            printf("exec() failed!\n");
        }
        perror("dup2");
    }
    return 0;
}
```

Παράδειγμα με exec & pipe & dups, II

- ❖ Το pipe4.c (μετάφραση με: gcc -o pipe4 pipe4.c):

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

int main() {
    int n;
    char msg[10];

    printf("(%d) - I am pipe4. I will read standard input.\n", getpid());

    while ((n = read(STDIN_FILENO, msg, 2)) >= 0) {
        if (n == 0) break; /* EOF */
        printf("(pipe4) - read %d bytes [%.*s]\n", n, n, msg);
    }
    if (n < 0) {
        perror("read() from child");
        exit(1);
    }
    return 0;
}
```

Εκτέλεση παραδείγματος

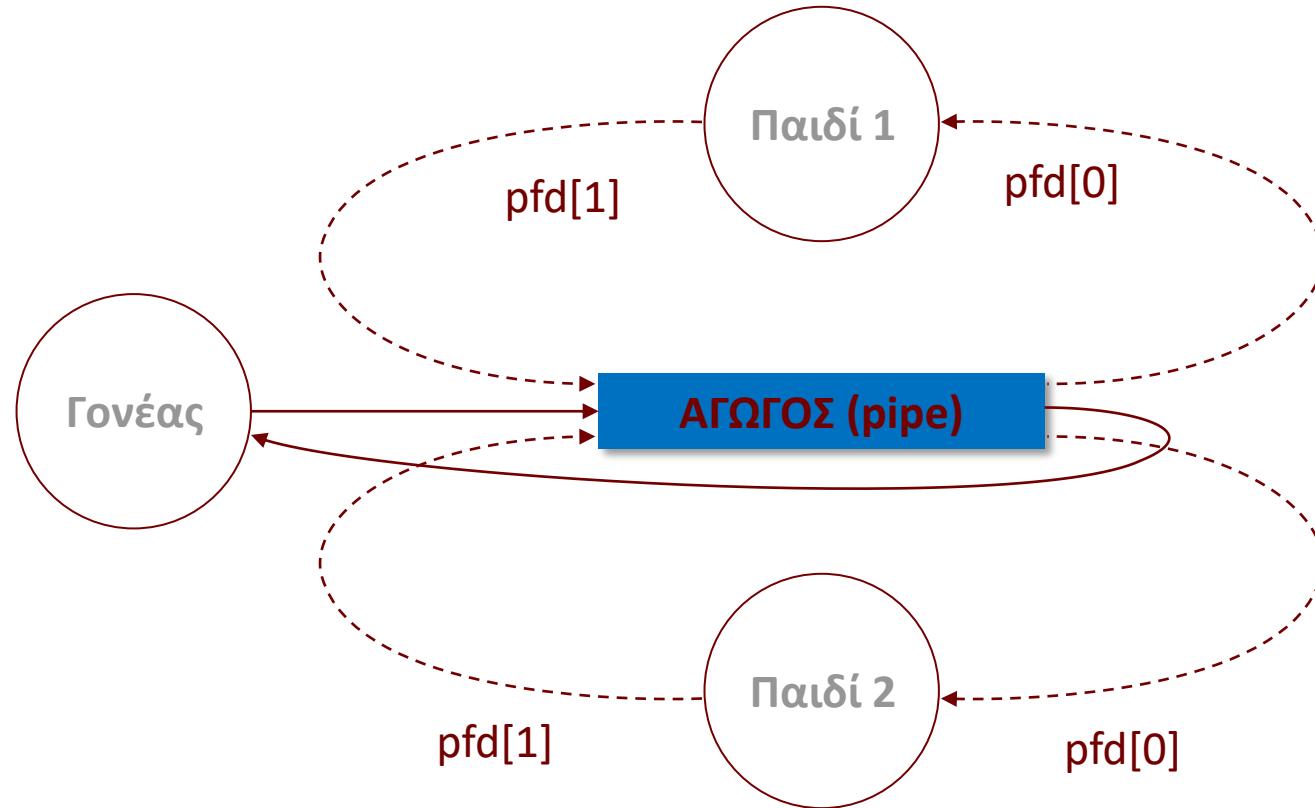
```
$ ./a.out
(24365) - I am the parent; pfd[0]=3, pdf[1]=4.
(24365) - I am the parent; sending hello message.
(24366) - I am the child. I will exec pipe4.
(24366) - I am pipe4. I will read standard input.
(pipe4) - read 2 bytes [he]
(pipe4) - read 2 bytes [ll]
(pipe4) - read 2 bytes [o!]
(pipe4) - read 1 bytes [!]
```



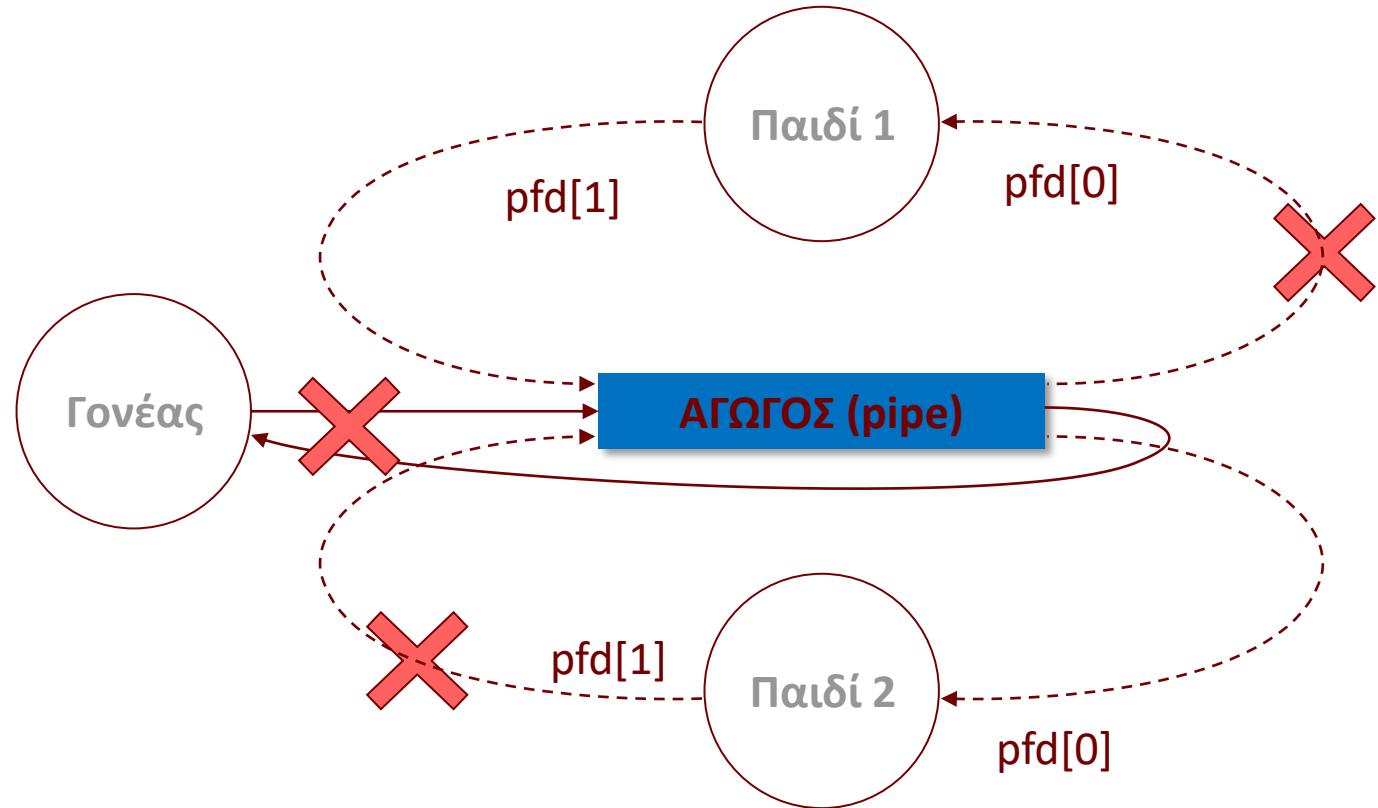
Επιπλέον παράδειγμα: 2 παιδιά

- ❖ Δημιουργήστε πρόγραμμα όπου η γονική διεργασία δημιουργεί **δύο (2)** παιδιά, όπου το πρώτο εκτελεί την εντολή “ls -l” και, μέσω αγωγού, στέλνει την έξοδο της εντολής στο δεύτερο παιδί, το οποίο εκτελεί την εντολή “wc”.
- ❖ Λύση:
 - Ο πατέρας θα δημιουργήσει έναν αγωγό
 - Στη συνέχεια θα δημιουργήσει 2 παιδιά
 - Τα παιδιά θα αντιγράψουν τα κατάλληλα άκρα του αγωγού με dup2()
στο stdin/stdout
 - Τα παιδιά θα κάνουν exec.

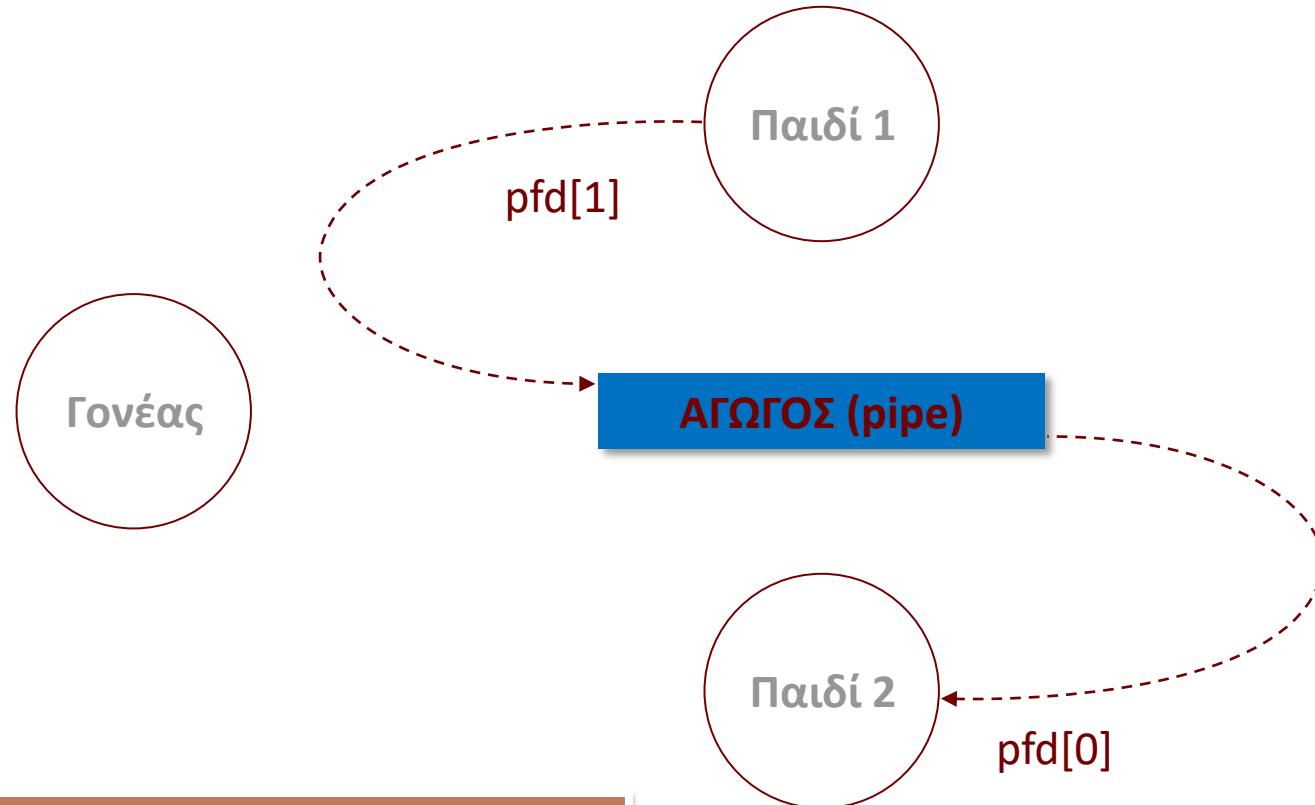
Σχηματισμός αγωγού



Σχηματισμός αγωγού



Σχηματισμός αγωγού



Υπενθύμιση:

Θα πείραζε αν ΔΕΝ κλείναμε τους περιγραφείς
και τους αφήναμε όλους ανοιχτούς;



Ο κώδικας

```
int main() {
    int pfd[2];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }

    if (fork() == 0) {      /* child 1 */
        close(pfd[0]);      /* No reading from the pipe */
        if (dup2(pfd[1], STDOUT_FILENO) >= 0) { /* Dup to std output */
            close(pfd[1]);          /* No longer needed! */
            execl("/bin/ls", "ls", "-l", NULL);
            printf("exec() failed from child 1!\n");
        }
        perror("dup2 from child 1");
        exit(1);
    }

    if (fork() == 0) {      /* child 2 */
        close(pfd[1]);      /* No writing to the pipe */
        if (dup2(pfd[0], STDIN_FILENO) >= 0) { /* Dup to std input */
            close(pfd[0]);          /* No longer needed! */
            execl("/usr/bin/wc", "wc", NULL);
            printf("exec() failed from child 2!\n");
        }
        perror("dup2 from child 2");
        exit(1);
    }
    ...
    return 0;
}

/* Parent code */
```

Είναι κάτι που πρέπει να κάνει ο γονέας αν θέλει να περιμένει να τελειώσουν τα παιδιά του;

Προγραμματισμός συστημάτων UNIX/POSIX

Προχωρημένη διαδιεργασιακή επικοινωνία:
επώνυμοι αγωγοί (FIFOs)
ουρές μηνυμάτων (message queues)
κοινόχρηστη μνήμη (shared memory)



MYY502

- ❖ Ότι είδαμε μέχρι τώρα μπορεί να χρησιμοποιηθεί για επικοινωνία διεργασιών που έχουν σχέση πατέρα – παιδιού
 - Διότι κληρονομούνται οι περιγραφείς αρχείων.
- ❖ Είναι περιοριστικό όμως. Μπορούμε να κάνουμε δύο ανεξάρτητες διεργασίες να επικοινωνήσουν;;;
- ❖ Απάντηση: Ναι! Και μάλιστα υπάρχουν πολλοί διαφορετικοί μηχανισμοί για το σκοπό αυτό.
 - «Προχωρημένη» διαδιεργασιακή επικοινωνία (interprocess communication – IPC).
 - Δεν υποστηρίζονται πάντα όλες οι διαφορετικές εκδόσεις των μηχανισμών σε όλα τα συστήματα



- ❖ **Επώνυμοι αγωγοί** (named pipes ή FIFOs)
- ❖ **Ουρές μηνυμάτων** (message queues)
 - Δύο εκδόσεις: System V (SysV) και POSIX
- ❖ **Κοινόχρηστη μνήμη** (shared memory)
 - Δύο εκδόσεις: System V (SysV) και POSIX
- ❖ **Υποδοχές** (sockets)
- ❖ **κλπ.**
 - Όλοι οι μηχανισμοί αφορούν διεργασίες οι οποίες εκτελούνται στο ίδιο μηχάνημα
 - Οι υποδοχές είναι ο μοναδικός μηχανισμός που *επιπρόσθετα επιτρέπει και την επικοινωνία διεργασιών οι οποίες εκτελούνται σε διαφορετικά συστήματα.*

FIFOs

- ❖ Πρόκειται για ένα «ειδικό» δυαδικό αρχείο το οποίο μπορούν να το ανοίξουν για διάβασμα ή γράψιμο δύο ανεξάρτητες διεργασίες αρκεί να ξέρουν το όνομά του (διαδρομή).
- ❖ Δημιουργία ενός FIFO:

```
int mkfifo(char *fullpath, mode_t permissions);
```

όπου τα permissions είναι οι άδειες, ίδιες με αυτές που δίνονται και κατά τη δημιουργία ενός νέου δυαδικού αρχείου στο τρίτο όρισμα της open().

- ❖ Αφού δημιουργηθεί το FIFO από κάποια διεργασία με την mkfifo(), οι εμπλεκόμενες διεργασίες (ακόμα και αυτή που το δημιούργησε) **πρέπει να το ανοίξουν με την open()**.
 - Αν δεν έχει δημιουργηθεί το FIFO, οι open() θα αποτύχουν.
- ❖ Από κει και μετά, λειτουργούν όπως οι αγωγοί με χρήση των write και read.
- ❖ Κατά την open(), η διεργασία που ανοίγει το FIFO για εγγραφή μπλοκάρει μέχρι μία άλλη διεργασία να κάνει open() για ανάγνωση και το αντίστροφο.
 - Όμως, αν απαιτείται, η διεργασία που ανοίγει για ανάγνωση μπορεί κατά την open() να δώσει το ειδικό flag O_NONBLOCK ώστε να μην μπλοκάρει, π.χ. O_RDONLY | O_NONBLOCK.

Παράδειγμα με FIFO: πελάτης

Πελάτης – client.c

```
#include "fifo.h"
int main() {
    int readfd, writefd;
    /* Open the FIFOs. We assume the server has already created them */
    if ( (writefd = open(FIFO1, O_WRONLY)) < 0)
        printf("client: can't open write fifo: %s", FIFO1);
    if ( (readfd = open(FIFO2, O_RDONLY)) < 0)
        printf("client: can't open read fifo: %s", FIFO2);

<... Κώδικας του πελάτη ...>

    close(readfd);
    close(writefd);
    /* Delete the FIFOs, now that we're finished. */
    if (unlink(FIFO1) < 0)
        printf("client: can't unlink %s", FIFO1);
    if (unlink(FIFO2) < 0)
        printf("client: can't unlink %s", FIFO2);
    return 0;
}
```

fifo.h

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS S_IRUSR | S_IWUSR
```

Παράδειγμα με FIFO: εξυπηρετητής

Εξυπηρετητής – server.c

```
#include "fifo.h"
int main() {
    int readfd, writefd;
    /* Create the FIFOs -- one for reading and one for writing. */
    if (mkfifo(FIFO1, PERMS) < 0)
        printf("can't create fifo: %s", FIFO1);
    if (mkfifo(FIFO2, PERMS) < 0) {
        unlink(FIFO1);
        printf("can't create fifo: %s", FIFO2);
    }
    /* Open the FIFOs - one for reading and one for writing */
    if ( (readfd = open(FIFO1, O_RDONLY)) < 0)                      /* A */
        printf("server: can't open read fifo: %s", FIFO1);
    if ( (writefd = open(FIFO2, O_WRONLY)) < 0)                         /* B */
        printf("server: can't open write fifo: %s", FIFO2);

<... Κώδικας του εξυπηρετητή ...>

    close(readfd);
    close(writefd);
    return 0;
}
```

Τι θα γίνει αν εναλλάξουμε τις δύο γραμμές Α και Β ?

fifo.h

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS S_IRUSR | S_IWUSR
```

- ❖ Οι μηχανισμοί διαδιεργασιακής επικοινωνίας SysV
 - a. Απαιτούν ένα **μοναδικό «κλειδί»** (συνήθως ακέραιος αριθμός)
 - b. Δεν τους χειριζόμαστε όπως τα δυαδικά αρχεία (δεν διαθέτουν περιγραφέα αρχείου)
 - c. Αφήνουν «υπολείμματα» κατά τον τερματισμό μίας εφαρμογής και για αυτό το λόγο πρέπει ο προγραμματιστής να τα «διαγράφει» ρητά, αλλιώς ξεμένουν και γεμίζουν τη μνήμη του συστήματος.
- ❖ Για τη δημιουργία του κλειδιού
 - είτε διαλέγουμε έναν «περίεργο» αριθμό που θεωρούμε ότι δεν τον χρησιμοποιούν άλλες διεργασίες
 - είτε, για μεγαλύτερη ασφάλεια, χρησιμοποιούμε τη συνάρτηση `ftok()`, η οποία δημιουργεί κλειδιά συνδυάζοντας μια υπάρχουσα διαδρομή αρχείου και έναν ακέραιο (όποιος την καλέσει με τα ίδια ορίσματα, παίρνει το ίδιο κλειδί).

```
#include <sys/ipc.h>
key_t ftok(char *fullpath, int id);
```
 - Η διαδρομή πρέπει να είναι οποιοδήποτε υπάρχον αρχείο ή κατάλογος και ο αριθμός `id` οποιοσδήποτε ακέραιος (όμως μόνο το 1^o byte λαμβάνεται υπόψη και πρέπει να είναι μη-μηδενικό), π.χ. `x = ftok("/tmp", 1);`
 - Για να εξασφαλίζεται 100% η μοναδικότητα του κλειδιού, καλό είναι η κάθε διαδρομή να δημιουργεί ένα δικό της προσωρινό αρχείο.

Ουρές μηνυμάτων (SysV message queues)

- ❖ Οι αγωγοί και τα FIFOs είναι μηχανισμοί επικοινωνίας ως μία ροή από bytes.
- ❖ Οι ουρές μηνυμάτων είναι μία λίστα από *structs*, καθένα από τα οποία περιέχει ένα διαφορετικό «μήνυμα» από μία διεργασία προς μία άλλη. Τη λίστα τη χειρίζεται το λειτουργικό σύστημα.
- ❖ Τα μηνύματα μπορεί να τα διαβάσει μία διεργασία με όποια σειρά θέλει.
- ❖ Οι ουρές μηνυμάτων είναι **δικατευθυντήριες** (δηλ. μπορεί να γράψουν και να διαβάσουν και οι δύο διεργασίες).



Κλήσεις για ουρές μηνυμάτων

```
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgget(key_t key, int msgflag);
```

- ❖ Δημιουργία και άνοιγμα ουράς μηνυμάτων με κλειδί το key. Το msgflag είναι συνήθως IPC_CREAT (όχι O_CREAT!). Επιστρέφει ένα περιγραφέα (id) για την ουρά.

```
int msgsnd(int msgqid, void *ptr, int length, int flag);
```

- ❖ Αποστολή-προσθήκη μηνύματος στην ουρά μηνυμάτων. Ο δείκτης ptr πρέπει να δείχνει σε ένα struct το οποίο έχει 2 πεδία: α) έναν long int που αντιπροσωπεύει τον «τύπο» του μηνύματος και β) τα δεδομένα του μηνύματος.

```
int msgrcv(int msgqid, void *ptr, int length, long msgtype, int flag);
```

- ❖ Παραλαβή-αφαίρεση μηνύματος από την ουρά μηνυμάτων. Παραλαμβάνει μόνο μήνυμα του δεδομένου τύπου (ή όποιο μήνυμα είναι πρώτο στην ουρά αν το msgtype είναι 0)

```
int msgctl(int msgqid, int cmd, struct msgqid_ds *buff);
```

- ❖ Χρησιμοποιείται για έλεγχο της ουράς (π.χ. διαγραφή της).

Κοινόχρηστη μνήμη (SysV shared memory)

- ❖ Εντελώς διαφορετικός μηχανισμός και συνήθως ο ταχύτερος.
- ❖ Χοντρικά κάποια διεργασία δεσμεύει μνήμη (κάτι σαν malloc) την οποία όμως την προσπελαύνουν πολλές διεργασίες!
- ❖ Όποια διεργασία τροποποιήσει κάτι, οποιαδήποτε άλλη μπορεί να δει την αλλαγή.
- ❖ Μπαίνουν θέματα **ταυτοχρονισμού / παραλληλισμού**. Τι γίνεται αν π.χ. δύο ή παραπάνω διεργασίες πάνε να τροποποιήσουν το ίδιο byte???
- ❖ Συνήθως συνδυάζονται με σηματοφόρους (**semaphores**) για τον έλεγχο του παραλληλισμού και το συγχρονισμό των διεργασιών.

Κλήσεις για κοινόχρηστη μνήμη

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int msgflag);
```

- ❖ Δέσμευση τμήματος μνήμης (κάτι σαν «malloc») με κλειδί το key. Το msgflag είναι συνήθως IPC_CREAT (όχι O_CREAT!). Επιστρέφει έναν περιγραφέα (id) για το τμήμα μνήμης.

```
void *shmat(int shmid, void *ptr, int flag);
```

- ❖ Η shmget() δεν επιστρέφει διεύθυνση μνήμης όπως η malloc(). Αυτό το κάνει η shmat() η οποία και «προσαρτεί» τον κοινόχρηστο χώρο μνήμης με το δεδομένο id στην τρέχουσα διεργασία. Το δεύτερο όρισμα είναι συνήθως NULL και το τρίτο 0. Εφόσον όλες οι διεργασίες εκτελέσουν την shmat(), ότι γράφει κάποια μέσα στο χώρο αυτό επηρεάζει όλες.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buff);
```

- ❖ Χρησιμοποιείται για έλεγχο της ουράς (π.χ. διαγραφή της).

Προγραμματισμός συστημάτων UNIX/POSIX

Σήματα (signals)



MYY502

Σήματα (signals)

- ❖ Τα σήματα είναι «διακοπές» λογισμικού (software interrupts) οι οποίες διακόπτουν την κανονική λειτουργία μίας διεργασίας.
- ❖ Προκαλούνται από διάφορες συνθήκες, π.χ. κάποιο πρόβλημα στο hardware, κάποια παράνομη λειτουργία (π.χ. διαίρεση δια μηδέν ή προσπέλαση σε θέση μνήμης που δεν επιτρέπεται) ή και από τον χρήστη (π.χ. πάτημα CTRL-C).
- ❖ Τα περισσότερα σήματα προκαλούν τον άμεσο τερματισμό της διεργασίας, εκτός και αν έχει προβλεφτεί τρόπος «διαχείρισής» τους από την ίδια την εφαρμογή.
- ❖ Υπάρχουν πολλά διαφορετικά είδη σημάτων, π.χ.

`SIGINT 2 /* interrupt */`, `SIGQUIT 3 /* quit */`,

`SIGILL 4 /* illegal instruction */`, `SIGKILL 9 /* hard kill */`,

`SIGALRM 14 /* alarm clock */`, `SIGCHLD 20 /* to parent on child exit */`



Οι «φάσεις» ενός σήματος

- ❖ **Signal generation**
Εφόσον συμβεί κάποιο γεγονός (π.χ. διαίρεση δια 0) τότε **παράγεται** ένα σχετικό σήμα (π.χ. SIGFPE)
- ❖ **Signal delivery**
Το σήμα **παραδίδεται** στη διεργασία που πρέπει
- ❖ **Signal action**
Με την παράδοση, προκαλείται κάποια **ενέργεια** (π.χ. τερματίζει τη διεργασία)
- ❖ **Pending signal**
Μεταξύ της στιγμής που παράγεται και της στιγμής που παραδίδεται στη διεργασία, το σήμα βρίσκεται σε φάση **εκκρεμότητας** (συνήθως αμελητέος χρόνος).
- ❖ Τα μόνα πράγματα που μπορεί να κάνει μία διεργασία είναι:
 - Να αλλάξει την ενέργεια (action) που προκαλείται από το σήμα
 - Να κρατάει ένα σήμα σε κατάσταση εκκρεμότητας επ' αόριστο, «**μπλοκάροντάς**» το (**blocked**). Το σήμα θα παραδοθεί όταν και εφόσον η διεργασία το «**ξεμπλοκάρει**» (**unblocked**)
 - ... κι αυτά, όχι για όλους τους τύπους σημάτων.



Παραδείγματα από σήματα

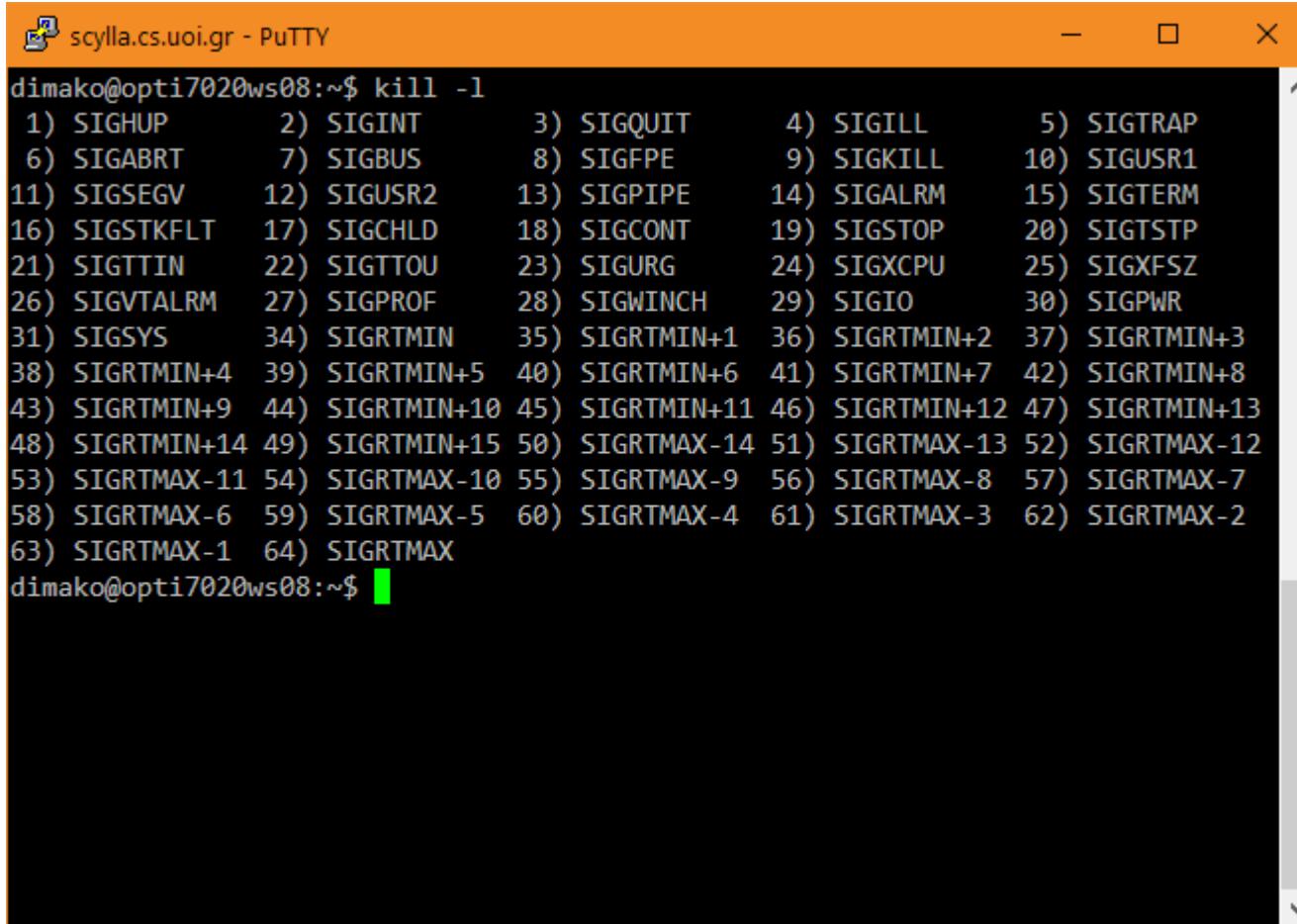
ΑΙΤΙΟ	ΣΗΜΑ	DEFAULT ACTION	ΜΠΛΟΚΑΡΕΤΑΙ;	ΑΛΛΑΖΕΙ Η ΕΝΕΡΓΕΙΑ;
Αριθμητικό σφάλμα (διαίρεση δια 0, υπερχείλιση κλπ)	SIGFPE	Τερματισμός διεργασίας	ΝΑΙ	ΝΑΙ
Πρόσβαση σε μνήμη που δεν κατέχει	SIGSEGV	Τερματισμός διεργασίας	ΝΑΙ	ΝΑΙ
Ο χρήστης πάτησε CTRL-C	SIGINT	Τερματισμός διεργασίας	ΝΑΙ	ΝΑΙ
\$ kill -9	SIGKILL	Τερματισμός διεργασίας	ΟΧΙ	ΟΧΙ
Τερματίζει μία Θυγατρική διεργασία	SIGCHLD	Αγνοείται	ΝΑΙ	ΝΑΙ

❖ Για περισσότερα:

\$ man signal ή
\$ man 7 signal

Μια λίστα από τα διαθέσιμα σήματα

- ❖ Τα κλασικά και συνηθισμένα σήματα είναι μέχρι το 31 (SIGSYS).
- ❖ Από το 32 και μετά είναι τα λεγόμενα σήματα πραγματικού χρόνου (Real Time Signals) που δεν μας αφορούν εδώ.



The screenshot shows a PuTTY terminal window titled "scylla.cs.uoi.gr - PuTTY". The command "kill -l" is run, displaying a list of signals. The list is organized into four columns and two continuation lines. The first column contains signal numbers 1 through 63. The second column contains signal names: SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGBUS, SIGFPE, SIGKILL, SIGUSR1, SIGSEGV, SIGUSR2, SIGPIPE, SIGALRM, SIGTERM, SIGSTKFLT, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, SIGURG, SIGXCPU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGWINCH, SIGIO, SIGPWR, SIGSYS, SIGRTMIN, SIGRTMIN+1, SIGRTMIN+2, SIGRTMIN+3, SIGRTMIN+4, SIGRTMIN+5, SIGRTMIN+6, SIGRTMIN+7, SIGRTMIN+8, SIGRTMIN+9, SIGRTMIN+10, SIGRTMIN+11, SIGRTMIN+12, SIGRTMIN+13, SIGRTMIN+14, SIGRTMIN+15, SIGRTMAX-14, SIGRTMAX-13, SIGRTMAX-12, SIGRTMAX-11, SIGRTMAX-10, SIGRTMAX-9, SIGRTMAX-8, SIGRTMAX-7, SIGRTMAX-6, SIGRTMAX-5, SIGRTMAX-4, SIGRTMAX-3, SIGRTMAX-2, SIGRTMAX-1, SIGRTMAX, and SIGRTMAX. The third and fourth columns are empty. The prompt "dimako@opti7020ws08:~\$ " is visible at the bottom.

```
dimako@opti7020ws08:~$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Πώς μπλοκάρω/ξεμπλοκάρω ένα σήμα;

- ❖ Κάθε διεργασία διαθέτει ένα σύνολο από σήματα που μπλοκάρει (και άρα δεν επιτρέπει στο σύστημα να της παραδίδονται), το λεγόμενο *signal mask*.
 - Αν παραχθεί κάποιο από τα σήματα που είναι στο signal mask, δεν θα της παραδοθεί και άρα δεν θα προκληθεί καμία ενέργεια από αυτό.
 - Το σήμα είναι pending. Θα παραδοθεί όταν (αν) η διεργασία το ξεμπλοκάρει.
- ❖ Για να μπλοκάρουμε ένα σήμα, αρκεί να το βάλουμε στο signal mask και για να το ξεμπλοκάρουμε αρκεί να το βγάλουμε από αυτό το σύνολο.
- ❖ Το signal mask αλλάζει με τη συνάρτηση: [επιστρέφει < 0 αν αποτύχει]

```
#include <signal.h>
int sigprocmask(int how, sigset_t *newmask, sigset_t *oldmask);
```

όπου το newmask είναι το νέο σύνολο, ενώ στο oldmask θα επιστραφεί το παλαιό (αν oldmask≠NULL). Το how καθορίζει πώς θα γίνει η αλλαγή:

- SET_SIGMASK: το νέο signal mask είναι ακριβώς αυτό που δίνεται στο newmask
- SET_SIGBLOCK: πρόσθεσε στο υπάρχον signal mask ότι έχει το newmask
- SET_SIGUNBLOCK: αφαίρεσε από το υπάρχον mask ότι έχει το newmask

sigprocmask() – συνέχεια

- ❖ Το `sigset_t` είναι μία δομή που καταγράφει ποια σήματα μπλοκάρονται.
- ❖ Οι συναρτήσεις που μπορούν να προσθαφαιρέσουν σήματα στη δομή αυτή είναι οι εξής:

<code>sigemptyset(sigset_t *set)</code>	Άδειασε το σύνολο. Κανένα σήμα δεν θα μπλοκαριστεί.
<code>sigfillset(sigset_t *set)</code>	Βάλε όλα τα σήματα μέσα στο σύνολο. Όλα μπλοκάρονται.
<code>sigaddset(sigset_t *set, int sigid)</code>	Πρόσθεσε το σήμα <code>sigid</code> στο σύνολο (θα μπλοκάρεται)
<code>sigdelset(sigset_t *set, int sigid)</code>	Αφαίρεσε το σήμα <code>sigid</code> από το σύνολο (ξεμπλοκάρεται)
<code>sigismember(sigset_t *set, int sigid)</code>	Έλεγχε αν το σήμα <code>sigid</code> είναι στο σύνολο



Παράδειγμα χρήσης

- ❖ Έλεγχος αν η τρέχουσα signal mask περιέχει το SIGINT και αν ναι, το ξεμπλοκάρουμε:

```
#include <signal.h>
...
int main() {
    sigset_t newmask, oldmask;

    ...a

    sigprocmask(SIG_SETMASK, NULL, &oldmask);      /* Just get the current mask */
    if (sigismember(&oldmask, SIGINT)) {             /* Check if SIGINT is blocked */
        sigemptyset(&newmask);                      /* Create an empty set */
        sigaddset(&newmask, SIGINT);                  /* Add the SIGINT signal */
        sigprocmask(SIG_UNBLOCK, &newmask, NULL);     /* Remove from current mask */
    }
    ...
}
```

Πώς αλλάζω την ενέργεια που θα προκληθεί;

- ❖ Προκειμένου να προσδιορίσουμε την ενέργεια που θα προκληθεί όταν και αν μας παραδοθεί κάποιο σήμα, θα πρέπει να χρησιμοποιήσουμε τη συνάρτηση `sigaction()`: [επιστρέφει < 0 αν αποτύχει]

```
#include <signal.h>
int sigaction(int sigid,
              struct sigaction *newact,
              struct sigaction *oldact);
```

όπου το `sigid` είναι το σήμα που μας ενδιαφέρει, και `newact` είναι μία δομή που περιγράφει τη νέα ενέργεια που θέλουμε να προκαλείται. Εφόσον το `oldact` δεν είναι `NULL`, κατά την επιστροφή θα περιέχει την περιγραφή της παλιάς ενέργειας.

- ❖ Οι ενέργειες που μπορούμε να ορίσουμε (μέσω του `newact`) είναι μία από τις παρακάτω τρεις:
 - Να αγνοηθεί το σήμα (και άρα να μην προκαλέσει τίποτε)
 - Να γίνει η ενέργεια που έχει εξ ορισμού το σύστημα (default action), όποια και να είναι αυτή
 - Να κληθεί μία δική μας συνάρτηση, κάτι που είναι γνωστό ως διαχειριστής σήματος (*signal handler*)

Καθορίζοντας την ενέργεια

Η δομή που περιγράφει την ενέργεια έχει ως εξής:

```
struct sigaction {  
    void (*sa_handler)(int); /* SIG_DFL, SIG_IGN ή handler δικό μας */  
    sigset_t sa_mask;        /* Επιπρόσθετα σήματα προς μπλοκάρισμα */  
    int sa_flags;            /* 0 συνήθως */  
    ...  
};
```

Λεπτομέρειες:

1. Το πεδίο `sa_handler` είναι δείκτης σε συνάρτηση δική μας (διαχειριστής σήματος) ή του δίνουμε την τιμή `SIG_DFL` (ώστε να εκτελεστεί η default action του συστήματος) ή του δίνουμε την τιμή `SIG_IGN` (ώστε να αγνοηθεί το σήμα).
2. Η συνάρτηση που ορίζουμε ως διαχειριστή θα έχει ως μοναδικό όρισμα την ταυτότητα του σήματος.

Καθορίζοντας την ενέργεια (συνέχεια)

Η δομή που περιγράφει την ενέργεια έχει ως εξής:

```
struct sigaction {  
    void (*sa_handler)(int); /* SIG_DFL, SIG_IGN ή handler δικό μας */  
    sigset_t sa_mask;        /* Επιπρόσθετα σήματα προς μπλοκάρισμα */  
    int sa_flags;            /* 0 συνήθως */  
    ...  
};
```

Λεπτομέρειες (συνέχεια):

3. Εφόσον εκτελεστεί η ενέργεια (π.χ. ο δικός μας handler), το σήμα που μας ενδιαφέρει θα είναι μπλοκαρισμένο μέχρι την ολοκλήρωσή της, οπότε και το σήμα θα ξεμπλοκαριστεί αυτόματα.
4. Αν μας ενδιαφέρει, προσωρινά, όσο εκτελείται η ενέργεια, να μπλοκαριστούν επιπρόσθετα σήματα, αυτά πρέπει να τα συμπεριλάβουμε στη μάσκα `sa_mask`. Με το τέλος της ενέργειας, αυτά τα σήματα ξεμπλοκάρονται αυτόματα.

Παράδειγμα διαχειριστή σήματος

- ❖ Η παρακάτω εφαρμογή δεν μπορεί να τερματιστεί με CTRL-C διότι έχει ορίσει ένα χειριστή για το σήμα SIGINT:

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

void handlectrlc(int sigid) {
    printf("You cannot stop me with CTRL-C!\n");
}

int main() {
    struct sigaction sact;

    sact.sa_handler = handlectrlc; /* Our handler to catch CTRL-C */
    sigemptyset(&sact.sa_mask);    /* No other signal to block */
    sact.sa_flags = 0;             /* Nothing special, must be 0 */
    if (sigaction(SIGINT, &sact, NULL) < 0)
        perror("could not set action for SIGINT");

    /* Loop for ever */
    while(1) {
        sleep(1);    /* Let it sleep for a while */
    }
    return 0;
}
```

Σημειώσεις (I): η συνάρτηση signal()

- ❖ Ο ορισμός ενός διαχειριστή σήματος γινόταν με πιο απλό (αλλά πλέον ξεπερασμένο τρόπο) με την κλήση `signal()`.

```
void (*signal(int sig, void (*func)(int)))(int);
```

Η οποία βασικά λέει στη `signal()` ότι αν συμβεί το σήμα `sig`, θα πρέπει να κληθεί η συνάρτηση `func`. Αν όλα πάνε καλά η `signal()` επιστρέφει δείκτη προς τον παλαιό handler, αλλιώς `SIG_ERR`.

- Η `signal` πρέπει να αποφεύγεται και να χρησιμοποιείται η `sigaction`, έστω κι αν φαίνεται πιο πολύπλοκη.
- ❖ Μέσα στον διαχειριστή, πρέπει να αποφεύγονται «επικίνδυνες» κλήσεις συστήματος (ακόμα και η `printf()` πρέπει να αποφεύγεται) και γενικότερα τα πράγματα πρέπει να γίνονται πολύ προσεκτικά – Ψάξτε να μελετήσετε τις λεπτομέρειες και τους λόγους για αυτό.

Σημειώσεις (II): σήματα ως διαδιεργασιακή επικοινωνία

- ❖ Μία διεργασία μπορεί να δημιουργήσει και να στείλει ένα σήμα σε μία άλλη (δεν επιτρέπονται όλα).

```
#include <signal.h>
int kill(pid_t pid, int signal);
```

- ❖ Υπάρχουν τα **SIGUSR1** και **SIGUSR2**.
- ❖ Με αυτό τον τρόπο, αν η διεργασία που το λαμβάνει έχει ορίσει τον αντίστοιχο διαχειριστή, είναι σαν να έχουμε «επικοινωνήσει» στέλνοντάς της έναν (συγκεκριμένο) ακέραιο αριθμό.
- ❖ Τα σήματα δεν μπορούν να χρησιμοποιηθούν για ανταλλαγή δεδομένων – το μόνο που μπορεί να «στείλει» ένα σήμα είναι ο εαυτός του και τίποτε άλλο.
- ❖ Περισσότερο για συγχρονισμό παρά επικοινωνία.

Προγραμματισμός συστημάτων UNIX/POSIX

*Χρονομέτρηση, καθυστέρηση και ανάλυση
επιδόσεων*



MYY502

Ανάγκη

- ❖ Πάρα πολύ συχνά υπάρχει η ανάγκη να χρονομετρήσουμε κάτι, π.χ.
 - Πόσο χρειάστηκε ένα πρόγραμμα για να εκτελεστεί
 - Πόσος χρόνος απαιτείται για συγκεκριμένα κομβικά τμήματα του κώδικα μας
 - Σύγκριση προγραμμάτων, benchmarking για επιδόσεις υλικού / λογισμικού
 - Profiling (στατιστικά χρονομέτρησης για να βρούμε που ένα πρόγραμμα καταναλώνει τον χρόνο του).
 - Η χρονομέτρηση δεν είναι κάτι απλό και χρειάζεται μεγάλη προσοχή.
- ❖ Ξεκινώντας από το τέλος (profiling), μπορούμε να βρούμε (μεταξύ άλλων) πόσο χρόνο σπαταλάει η εφαρμογή μας σε κάθε συνάρτηση.
- ❖ Το εργαλείο ονομάζεται **gprof** (GNU profiler)
 - Για τη χρήση του απαιτείται να δοθεί το flag `-pg` κατά τη μετάφραση με τον `gcc`.
 - Κατά την εκτέλεση του `a.out` (η οποία είναι αρκετά πιο αργή από ότι θα περιμέναμε) δημιουργείται ένα αρχείο "`gmon.out`".
 - Εκτελούμε το `gprof`, το οποίο με βάση το αρχείο αυτό μας δείχνει ενδιαφέρουσες πληροφορίες.



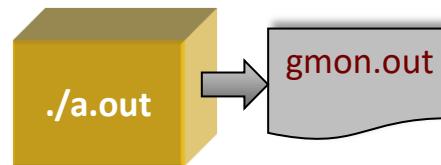
Profiling με το gprof

1. Μετάφραση με -pg



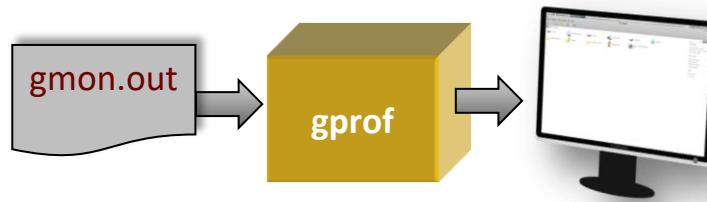
```
gcc -pg prog.c
```

2. Εκτέλεση προγράμματος



```
./a.out
```

3. Εκτέλεση gprof



```
gprof a.out gmon.out ...
```



Παράδειγμα

testprof.c

```
int a() {
    int i, sum=0;
    for (i = 0; i < 100000; i++)
        sum += i;
    return sum;
}
int b(void) {
    int i, sum=0;
    for (i = 0; i < 400000; i++) /* should be x4 the time of a() */
        sum += i;
    return sum;
}
int main() {
    int iterations = 1000;
    printf("profiling example.\n");
    for ( ; iterations > 0; iterations--) {
        a();
        b();
    }
    return 0;
}
```

ΤΕΡΜΑΤΙΚΟ

```
$ gcc -pg testprof.c
$ ./a.out
$
```

Προβολή πληροφοριών

- ❖ Μπορούμε να δούμε τα αποτελέσματα με 3 τρόπους:

1. Flat profile

Δείχνει πόσος χρόνος σπαταλήθηκε στην κάθε συνάρτηση και πόσες φορές έγινε κλήση στη συνάρτηση αυτή.

2. Call graph

Για κάθε συνάρτηση, δείχνει ποιες συναρτήσεις την κάλεσαν, ποιες κάλεσε αυτή και πόσες φορές έγιναν αυτά.

3. Annotated source

Δείχνει τον κώδικα και πόσες φορές εκτελέστηκαν διάφορα τμήματά του. Για τη συγκεκριμένη προβολή, πρέπει να έχουμε δώσει και το όρισμα $-g$ κατά τη μετάφραση.

Flat profile

Περίοδος "δειγματοληψίας".

Όποιο αποτέλεσμα είναι μικρότερο από αυτόν το χρόνο, θεωρείται ανακριβές.

```
$ gprof a.out gmon.out -p
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
81.38	0.83	0.83	1000	830.12	830.12	b
19.85	1.03	0.20	1000	202.47	202.47	a

Ποσοστό του χρόνου εκτέλεσης που σπαταλήθηκε στην κάθε συνάρτηση.

Πόσος χρόνος (sec) αφιερώθηκε στην κάθε συνάρτηση

Και οι δύο κλήθηκαν 1000 φορές

Χρόνος (μsec) για την κάθε κλήση

Call graph

```
$ gprof a.out gmon.out -q
```

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.97% of 1.03 seconds

	index	% time	self	children	called	name
[1]	100.0	0.00	1.03			<spontaneous>
				0.00	1000/1000	main [1] ← Η συνάρτηση
				0.20	1000/1000	b [2] ← Ακολουθούν a [3] ← αυτές που καλεί
[2]	80.4	0.83	0.83	0.00	1000/1000	main [1]
				0.00	1000	b [2]
[3]	19.6	0.20	0.20	0.00	1000/1000	main [1] ← Προηγούνται αυτές που την καλούν
				0.00	1000	a [3] ← Η συνάρτηση
...						

Πόσος χρόνος (sec) αφιερώθηκε
στις συναρτήσεις που κάλεσε (τις
ονομάζει "παιδιά" της)

Η printf() πού είναι;

Πρέπει όλες οι συναρτήσεις να
έχουν μεταφραστεί με -pg

Annotated source

```
$ gcc -pg -g testprof.c  
$ ./a.out  
$ gprof a.out gmon.out -A
```

```
*** File /home/dimako/testprof.c:  
      #include <stdio.h>  
1000 -> int a() {  
    int i, sum=0;  
    for (i = 0; i < 100000; i++)  
        sum += i;  
    return sum;  
}  
1000 -> int b(void) {  
    int i, sum=0;  
    for (i = 0; i < 400000; i++)  
        sum += i;  
    return sum;  
}  
##### -> int main() {  
    int iterations = 1000;  
    printf("profiling example\n");  
    for ( ; iterations > 0; iterations--) {  
        a();  
        b();  
    }  
}
```

...

Κάθε block του κώδικα πόσες φορές εκτελέστηκε

Χρονομέτρηση (I)

- ❖ Πρώτα από όλα, **τι ακριβώς θέλουμε να μετρήσουμε;**
 - Μας ενδιαφέρει ο χρόνος που ο επεξεργαστής αφιέρωσε σε ένα πρόγραμμα ή σε ένα συγκεκριμένο τμήμα του;
 - » **Χρόνος επεξεργαστή (χρόνος καθαρών υπολογισμών).**
 - Ή μας ενδιαφέρει πόση ώρα περνάει από ένα συγκεκριμένο γεγονός;
 - » **Πραγματικός χρόνος που παρήλθε.**
- ❖ Στην πρώτη περίπτωση, μας ενδιαφέρει μόνο πόσο χρόνο χρειάστηκε ο επεξεργαστής για καθαρούς υπολογισμούς, όχι πόσος χρόνος πέρασε από την ώρα που ξεκίνησε η χρονομέτρηση.
 - Ο χρόνος είναι ανεξάρτητος του φόρτου του συστήματος – θα είναι πάντα ο ίδιος, ακόμα και όταν εκτελούνται πολλές διεργασίες "ταυτόχρονα".
 - Μετρά "χρόνο επεξεργαστή" (CPU time) όχι πραγματικό χρόνο που παρήλθε (real/elapsed time).
 - Ίσως η πιο χρήσιμη περίπτωση όταν ενδιαφερόμαστε για εκτίμηση της επίδοσης του κώδικα.
- ❖ Στη δεύτερη περίπτωση μας ενδιαφέρει πόσος **πραγματικός χρόνος παρήλθε**.
 - *Εξαρτάται από τον φόρτο του συστήματος!* Όταν εκτελείται μόνο η εφαρμογή μας, η χρονομέτρηση θα είναι ακριβής, όταν εκτελούνται κι άλλες θα έχει διακυμάνσεις. Η χρονομέτρηση πρέπει να γίνεται σε ελεγχόμενο περιβάλλον.
 - Είναι χρήσιμο όταν μας ενδιαφέρει η συνολική συμπεριφορά, σε πραγματικές συνθήκες.



❖ Τι τάξη μεγέθους είναι οι χρόνοι που μας ενδιαφέρουν;

- Είναι χρόνοι της τάξης των λεπτών/ωρών/ημερών κλπ ;
 - ✧ Εδώ δεν μπαίνει θέμα ακρίβειας.
 - Είναι χρόνοι τάξης δευτερολέπτων / δεκάτων ή εκατοστών του δευτερολέπτου;
 - ✧ Κι εδώ η ακρίβεια δεν είναι (συνήθως) ουσιαστικό θέμα, οι περισσότεροι μηχανισμοί χρονομέτρησης είναι εφαρμόσιμοι.
 - Είναι χρόνοι της τάξης του msec / μsec / nsec?
 - ✧ Εδώ απαιτείται χρονομέτρηση με κατάλληλη υποστήριξη από το υλικό.
Οι χρονομετρητές πρέπει να έχουν αυξημένη ακρίβεια και ανάλυση (resolution).
-
- Στην τελευταία περίπτωση δεν υπάρχουν πάντα λύσεις τυποποιημένες που δουλεύουν σε όλα τα συστήματα.
 - Για τις υπόλοιπες περιπτώσεις μπορούν να χρησιμοποιηθούν αρκετοί και διαφορετικοί μηχανισμοί.

- ❖ **Τι μηχανισμούς χρονομέτρησης διαθέτουμε;**
 - Μετρούν χρόνο ("wall-clock" timers);
 - Μετρούν διαστήματα (interval timers);
- ❖ Στην πρώτη περίπτωση, οι χρόνοι που παίρνουμε είναι «έτοιμοι» για χρήση.
- ❖ Στη δεύτερη περίπτωση, θα πρέπει να ξέρουμε τη διάρκεια του κάθε διαστήματος προκειμένου να βρούμε τον πραγματικό χρόνο:
 - Π.χ. ρολόγια που αυξάνουν έναν μετρητή ανά τακτά διαστήματα.
 - Π.χ. μετρητές υψηλής ακρίβειας της CPU που μετρούν το πλήθος των παλμών του ρολογιού (clock cycles): θα πρέπει να γνωρίζουμε τη συχνότητα του επεξεργαστή για να βρούμε ποια είναι η διάρκεια του κάθε παλμού και άρα σε πόσο χρόνο αντιστοιχούν οι παλμοί που μετρήσαμε.

Δύο τρόποι χρονομέτρησης

1. Από το τερματικό

- Εντολή time: μετρά την εκτέλεση ενός ολόκληρου προγράμματος

```
$ time ./a.out
```

...

real	0m5.316s
user	0m2.304s
sys	0m0.004s

Real: πραγματικός χρόνος που παρήλθε (εμπεριέχει ότι καθυστερήσεις υπήρχαν, π.χ. αναμονή να πληκτρολογήσει κάτι ο χρήστης)

User: χρόνος καθαρών υπολογισμών (CPU time) χωρίς τις κλήσεις συστήματος

Sys: χρόνος καθαρών υπολογισμών (CPU time) που δαπανήθηκαν σε κλήσεις συστήματος

2. Προγραμματιστικά

- Για χρονομέτρηση ενός τμήματος του κώδικα μας:

```
<timing_call 1> /* Record time t1 */  
<κώδικας> /* The code we want to measure */  
<timing_call 2> /* Record time t2 */
```

- Η διαφορά των δύο χρόνων t2, t1 θα μας δώσει την επιθυμητή μέτρηση

Χρονομέτρηση με την `clock()`

- ❖ Η `clock()` μετρά καθαρό χρόνο εκτέλεσης (CPU time)
 - Συνήθως ακρίβεια εκατοστού του δευτερολέπτου
- ❖ Η `clock()` είναι interval-based και επιστρέφει τον χρόνο που αφιέρωσε η CPU από τη στιγμή που ξεκίνησε το πρόγραμμά σας, μετρημένο σε «κύκλους» (clocks).
 - Για να βρείτε το χρόνο σε δευτερόλεπτα θα πρέπει να διαιρέσετε με τη σταθερά `CLOCKS_PER_SEC`.
 - Προσοχή στη διαίρεση γιατί και το `CLOCKS_PER_SEC` και η τιμή επιστροφής της `clock()` είναι ακέραιοι.
- ❖ Θα πρέπει να κάνετε `#include <time.h>`.
- ❖ Προσοχή: επειδή η `clock()` επιστρέφει ακέραιο, αν το πρόγραμμά σας χρειάζεται πολλή ώρα να τρέξει υπάρχει κίνδυνος να μηδενιστεί ο χρονομετρητής και να λάβετε λάθος μετρήσεις. Π.χ. στο solaris αναφέρεται ότι ο χρονομετρητής μηδενίζεται και μετρά πάλι από την αρχή κάθε 36 λεπτά καθαρού χρόνου εκτέλεσης!

\$ man clock

13

Παράδειγμα χρονομέτρησης με την `clock()`

```
#include <stdio.h>
#include <time.h> /* Για την clock() */

int main() {
    double t1, t2; /* Για αποφυγή ακέραιας διαίρεσης */
    int i, sum = 0;

    t1 = (double) clock(); /* επιστρέφει clock_t (συνήθως int ή long) */
    for (i = 0; i < 1000000000; i++)
        sum += i;
    t2 = (double) clock();

    printf("Added 10000000 numbers in %lf sec (CPU time).\n",
           (t2 - t1) / CLOCKS_PER_SEC );
    return 0;
}
```

Χρονομέτρηση με την times()

- ❖ Η **times()** μετρά και πραγματικό χρόνο αλλά και καθαρό χρόνο εκτέλεσης (CPU time).
 - Ο πραγματικός χρόνος που επιστρέφει είναι το χρονικό διάστημα που παρήλθε από κάποιο απροσδιόριστο σημείο στο παρελθόν (π.χ. system boot time).
 - `#include <sys/times.h>`.
- ❖ Και η **times()** είναι interval-based. Όμως επιστρέφει χρόνους μετρημένους σε «χτύπους ρολογιού» (clock ticks).
 - Για να βρείτε το χρόνο σε δευτερόλεπτα θα πρέπει να διαιρέσετε με το πλήθος των χτύπων ρολογιού ανά δευτερόλεπτο, το οποίο το βρίσκεται μόνο προγραμματιστικά ως εξής:

```
ticspersec = sysconf(_SC_CLK_TCK); /* unistd.h */
```
 - Προσοχή πάλι στις διαιρέσεις γιατί και οι χτύποι ανά δευτερόλεπτο και η τιμή επιστροφής της **times()** είναι ακέραιοι.
- ❖ Επιστρέφει τον πραγματικό χρόνο που παρήλθε.
- ❖ Παίρνει ως παράμετρο ένα **struct tms** από όπου μπορούμε να μάθουμε για τους καθαρούς χρόνους εκτέλεσης:

```
struct tms {  
    clock_t tms_utime, tms_stime    /* for me */  
    clock_t tms_cutime, tms_cstime; /* for my child processes */  
};
```

\$ man -s 3 times

Παράδειγμα χρονομέτρησης με την times()

```
#include <stdio.h>
#include <sys/types.h>          /* Για την times() */
#include <unistd.h>             /* Για την sysconf() */

int main() {
    double t1, t2, cpu_time;      /* Για αποφυγή ακεραίας διαιρεσης */
    struct tms tb1, tb2;         /* Το χρειάζεται η times() */
    long   ticspersec;
    int    i, sum = 0;

    t1 = (double) times(&tb1); /* Η times() επιστρέφει (long) int */
    for (i = 0; i < 100000000; i++)
        sum += i;
    t2 = (double) times(&tb2);

    cpu_time = (double) ((tb2.tms_utime + tb2.tms_stime) -
                         (tb1.tms_utime + tb1.tms_stime));
    ticspersec = sysconf(_SC_CLK_TCK); /* # clock ticks / sec */

    printf("Real time: %lf sec; CPU time: %lf sec.\n",
           (t2 - t1) / ticspersec, cpu_time / ticspersec);
    return 0;
}
```

Χρονομέτρηση με την gettimeofday()

- ❖ Η **gettimeofday()** μετρά τον πραγματικό χρόνο που παρήλθε...
 - ... από την 1/1/1970, ώρα 00:00 (το λεγόμενο «Epoch»).
 - `#include <sys/time.h> /* Όχι το sys/times.h !! */`
 - Πολύ συχνή η χρήση της στην πράξη.
- ❖ Η **gettimeofday()** επιστρέφει χρόνο (wall-clock time).
 - Υλοποιείται συνήθως (όχι πάντα) με αρκετά μεγάλη ανάλυση (της τάξης του 1μsec).
- ❖ Παίρνει δύο παραμέτρους, με τη δεύτερη συνήθως NULL. Η πρώτη παράμετρος είναι δείκτης σε ένα **struct timeval** με τα εξής πεδία:

```
struct timeval {  
    time_t tv_sec;          /* seconds */  
    unsigned int tv_usec;    /* microseconds */  
};
```

To `time_t` ήταν μέχρι πρότινος ένας ακέραιος 32bit. Σε 68 χρόνια γίνεται overflow!
-- βλ. "year 2038 problem"

\$ man gettimeofday



Παράδειγμα χρονομέτρησης με την gettimeofday()

```
#include <stdio.h>
#include <sys/time.h>          /* Για την gettimeofday() */

int main() {
    struct timeval tv1, tv2;
    int i, sum = 0;
    double t;

    gettimeofday(&tv1, NULL);
    for (i = 0; i < 100000000; i++)
        sum += i;
    gettimeofday(&tv2, NULL);

    t = (tv2.tv_sec - tv1.tv_sec) +           /* seconds */
        (tv2.tv_usec - tv1.tv_usec)*1.0E-6; /* convert µsec to sec */

    printf("real time: %lf sec.\n", t);
    return 0;
}
```

Τεχνική: εύρεση της ανάλυσης της gettimeofday()

- ❖ Πώς μπορώ να βρω τι ανάλυση (resolution) έχει η gettimeofday();
 - Δηλαδή, ποιος είναι ο μικρότερος χρόνος που μπορεί να μετρήσει;

```
struct timeval tv1, tv2;  
int resolution;  
  
gettimeofday(&tv1, NULL);  
do {  
    gettimeofday(&tv2, NULL);  
}  
while (tv1.tv_usec == tv2.tv_usec);      /* Μέχρι να αλλάξει! */  
  
resolution = tv2.tv_usec - tv1.tv_usec; /* Σε μsec */
```

(θεωρώντας ότι ο χρόνος για την κλήση της gettimeofday() είναι αμελητέος σε σχέση με την ανάλυση)

Χρονομέτρηση με την `clock_gettime()`

- ❖ Η `clock_gettime()` είναι η πλέον σύγχρονη κλήση:
 - Μετρά με βάση κάποιο από τα παρεχόμενα **ρολόγια**.
 - Σε όλα τα συστήματα POSIX εγγυημένα υπάρχει ένα ρολόι που μετρά πραγματικό χρόνο (**CLOCK_REALTIME**).
 - Διάφορα συστήματα παρέχουν επιπλέον ρολόγια.
 - ✧ Π.χ. στο Solaris υπήρχε το **CLOCK_HIGHRES** (πραγματικού χρόνου με υπερυψηλή ανάλυση)
 - ✧ Σε πρόσφατες εκδόσεις του Linux υπάρχει το **CLOCK_PROCESS_CPUTIME_ID** (για χρόνους καθαρών υπολογισμών στη CPU).
 - `#include <time.h>`.

- ❖ Κλήση:

```
clock_gettime(clockid_t clk, struct timespec *tp);
```

```
struct timespec {  
    time_t tv_sec;           /* seconds */  
    long   tv_nsec;          /* nanoseconds */  
};
```

(στο `tv_nsec` μπορούμε να βάλουμε από 0 μέχρι 999.999.999).

- ❖ Επιπλέον ευκολία:

```
clock_getres(clockid_t clk, struct timespec *tp);
```

- Στο `tp` λαμβάνουμε την ανάλυση (resolution) του ρολογιού `clk`.

\$ man `clock_gettime`

20

Παράδειγμα χρονομέτρησης με την `clock_gettime()`

```
#include <stdio.h>
#include <time.h>          /* Για την clock_gettime() */

int main() {
    struct timespec ts1, ts2;
    int    i, sum = 0;
    double t;

    clock_gettime(CLOCK_REALTIME, &ts1);
    for (i = 0; i < 100000000; i++)
        sum += i;
    clock_gettime(CLOCK_REALTIME, &ts2);

    t = (ts2.tv_sec - ts1.tv_sec) +           /* seconds */
        (ts2.tv_nsec - ts1.tv_nsec)*1.0E-9;   /* convert nsec to sec */
    printf("real time: %lf sec.\n", t);

    clock_getres(CLOCK_REALTIME, &ts1);
    printf("clock resolution: %lf nsec.\n", ts1.tv_sec*1.0E9 + ts1.tv_nsec);
    return 0;
}
```

Καθυστέρηση – αναμονή (I)

- ❖ Μια πρακτική αναγκαιότητα είναι η τεχνητή καθυστέρηση.
 - Είτε ενδιαφερόμαστε να καθυστερήσουμε τα πρόγραμμά μας για λίγο (π.χ. για να προλάβει να γίνει κάποιο γεγονός)
 - Είτε θέλουμε να αφήσουμε να περάσει ένα προκαθορισμένο διάστημα προκειμένου να χρονομετρήσουμε κάτι.
- ❖ Και στις δύο περιπτώσεις, μπορούμε να πούμε στο σύστημα να "κοιμίσει" τη διεργασία μας για ένα συγκεκριμένο χρονικό διάστημα.
 - Η διεργασία σταματά προσωρινά να εκτελείται
 - Το σύστημα εκτελεί ότι άλλες διεργασίες έχει (για να μην κάθετε)
 - Όταν παρέλθει το χρονικό διάστημα που ορίσαμε, συνεχίζει την εκτέλεση της διεργασίας μας
- ❖ Συναρτήσεις τύπου "sleep()"

Καθυστέρηση – αναμονή (II)

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
int usleep(unsigned int microsecs); /* μsec */
```

- ❖ Η `sleep()` είναι η πιο κλασική κλήση, αλλά είναι για σχετικά μεγάλα διαστήματα ($\geq 1 \text{ sec}$)
- ❖ Η `usleep()` είναι για διαστήματα μέχρι 1 sec (δεν δουλεύει για μεγαλύτερα διαστήματα, το όρισμα πρέπει να είναι $\leq 1.000.000$).

```
#include <time.h>
int nanosleep(struct timespec *req, struct timespec *rem);
```

- ❖ Το `req` προδιορίζει το πλήθος των nanoseconds για το διάστημα αναμονής (όπως στην `clock_gettime()`)

```
struct timespec {
    time_t tv_sec;          /* seconds */
    long   tv_nsec;          /* nanoseconds */
};
```
- ❖ Ναι μεν προσδιορίζουμε `nsec`, αλλά αν το διάστημα αναμονής είναι μικρότερο από την ανάλυση του ρολογιού του συστήματος, ο χρόνος αναμονής θα είναι μεγαλύτερος από αυτόν που ζητήσαμε.
- ❖ Το `rem` είναι συνήθως `NULL`.

\$ man nanosleep

23

Χρονόμετρο – αντίστροφη μέτρηση

- ❖ Πολλές φορές υπάρχει η ανάγκη να γνωρίζουμε πότε παρέρχεται ένα συγκεκριμένο χρονικό διάστημα.
 - Παραδείγματα:
 - ✧ Θέλουμε να υλοποιήσουμε αντίστροφη μέτρηση
 - ✧ Θέλουμε κάθε 15 λεπτά να εκτυπώνεται ένα ενημερωτικό μήνυμα προς τον χρήστη.
 - ✧ Θέλουμε σε ένα παιχνίδι να μετρήσουμε πόσες φορές ο χρήστης πάτησε ένα πλήκτρο μέσα σε 10 δευτερόλεπτα.
 - ✧ κλπ
 - Κατά τη διάρκεια αυτού του διαστήματος, θέλουμε η εφαρμογή μας να συνεχίζει την εκτέλεσή της
 - Επομένως οι συναρτήσεις τύπου `sleep()` δεν μπορούν να χρησιμοποιηθούν
- ❖ Οι συναρτήσεις που παρέχονται για τέτοιες περιπτώσεις είναι οι συναρτήσεις «ξυπνητηριών»
 - `alarm()`
 - `setitimer()`
- ❖ Η βασική ιδέα πίσω από αυτές είναι:
 - Ενεργοποίηση ενός ξυπνητηριού / χρονομετρητή (`timer`) που μετράει αντίστροφα
 - Όταν παρέλθει το διθέν χρονικό διάστημα, προκαλείται διακοπή (`interrupt`) στη διεργασία
 - Έχουμε φροντίσει να έχουμε δική μας συνάρτηση για την εξυπηρέτηση της διακοπής

alarm()

- ❖ Η «κλασική» κλήση είναι η `alarm()`, η οποία όμως μπορεί να μετρήσει μόνο δευτερόλεπτα:

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
```

- ❖ Η παράμετρος `sec` καθορίζει το χρονικό διάστημα σε δευτερόλεπτα.
- ❖ Όταν ολοκληρωθεί το χρονικό διάστημα προκαλείται σήμα / διακοπή τύπου `SIGALRM`.
- ❖ Παρατηρήσεις:
 1. Αν η συνάρτηση κληθεί πριν τελειώσει το προηγούμενο χρονικό διάστημα που είχε τεθεί, ακυρώνεται το παλιό και ορίζεται νέο διάστημα.
 2. Καλώντας την με παράμετρο 0, **ακυρώνεται** το χρονόμετρο.
 3. Επιστρέφει το χρόνο που έμενε μέχρι να ολοκληρωθεί το προηγούμενο διάστημα.
 4. Επειδή και η `sleep()` μπορεί να υλοποιηθεί με χρήση των ίδιων χρονομέτρων με την `alarm()`, δεν πρέπει να γίνεται `sleep()` πριν την ολοκλήρωση του χρονικού διαστήματος από την `alarm()`.

setitimer() - I

- ❖ Η συνιστώμενη κλήση είναι η `setitimer()` :

```
#include <sys/time.h>
int setitimer(int which, /* ποιο χρονόμετρο να χρησιμοποιηθεί */
              struct itimerval *new,
              struct itimerval *old);
```

- ❖ Επιστρέφει 0 αν πέτυχε, ή < 0 σε περίπτωση αποτυχίας.
- ❖ Για κάθε διεργασία ορίζονται 3 διαφορετικά χρονόμετρα
 - `ITIMER_REAL`, για αντίστροφη χρονομέτρηση σε πραγματικό χρόνο.
 - `ITIMER_VIRTUAL`, για αντίστροφη χρονομέτρηση σε χρόνο εκτέλεσης (CPU user time)
 - `ITIMER_PROF`, για αντίστροφη χρονομέτρηση σε χρόνο εκτέλεσης που περιλαμβάνει και κλήσεις συστήματος (CPU user + system time, κυρίως για profiling και debugging).
 - Μόνο ένα χρονόμετρο μπορεί να έχετε ενεργό σε οποιαδήποτε χρονική στιγμή.
- ❖ Το `ITIMER_REAL` είναι το ίδιο με αυτό που χρησιμοποιεί και η `alarm()`, οπότε δεν πρέπει να μπλέκονται οι δύο κλήσεις.

setitimer() - II

```
int setitimer(int which,  
             struct itimerval *new,  
             struct itimerval *old);
```

- ❖ Το old είναι συνήθως NULL, αλλιώς εκεί επιστρέφεται ο χρόνος που απέμενε από το προηγούμενο χρονόμετρο.
- ❖ Το χρονικό διάστημα προσδιορίζεται στο new που έχει τύπο:

```
struct itimerval {  
    struct timeval it_interval; /* next value */  
    struct timeval it_value;   /* current value */  
};
```

Στο **it_value** δίνουμε το χρονικό διάστημα που πρέπει να παρέλθει. Όταν ολοκληρωθεί, **ΤΟ ΧΡΟΝΟΜΕΤΡΟ ΞΑΝΑΡΧΙΖΕΙ ΝΑ ΜΕΤΡΑΕΙ ΑΝΤΙΣΤΡΟΦΑ** για διάστημα ίσο με **it_interval**.

- Αν μας ενδιαφέρει μόνο μία χρονομέτρηση, θα πρέπει το **it_interval** να το θέσουμε σε **μηδενική** τιμή,
 - αλλιώς θα έχουμε συνεχώς ξυπνήματα κάθε **it_interval** χρόνο.
-
- ❖ Όταν ολοκληρωθεί το διάστημα, παράγεται διαφορετικό signal ανάλογα με το χρονόμετρο που χρησιμοποιήθηκε:
 - SIGALRM για το ITIMER_REAL.
 - SIGVTALRM για το ITIMER_VIRTUAL
 - SIGPROF για το ITIMER_PROF

To struct timeval είναι γνωστό από την gettimeofday():

```
struct timeval {  
    time_t tv_sec;  
    unsigned int tv_usec;  
};
```

Πιθανή υλοποίηση της alarm() μέσω setitimer()

```
unsigned int myalarm (unsigned int sec) {
    struct itimerval old, new;

    new.it_value.tv_sec      = (long int) sec;
    new.it_value.tv_usec     = 0;
    new.it_interval.tv_sec   = 0; /* do not repeat */
    new.it_interval.tv_usec = 0;
    if (setitimer(ITIMER_REAL, &new, &old) < 0)
        return 0;
    else
        return old.it_value.tv_sec;
}
```



Απλό παράδειγμα χρήσης

```
#include <stdio.h>      /* for printf */
#include <sys/time.h>    /* for setitimer */
#include <signal.h>       /* for signal */

void handleAlarm(int);

int main() {
    struct itimerval it_val;                      /* for setting itimer */

    /* sigaction() should actually be used - I use signal() due to lack of slides space */
    if (signal(SIGALRM, handleAlarm) == SIG_ERR) { /* Set SIGALRM handler */
        perror("Unable to catch SIGALRM");
        exit(1);
    }
    it_val.it_value.tv_sec = 0;                     /* Set the timer */
    it_val.it_value.tv_usec = 500000;                /* 0.5 sec */
    it_val.it_interval = it_val.it_value;           /* repeat every 0.5 sec */
    if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
        perror("error calling setitimer()");
        exit(1);
    }
    while (1)
        ;
    return (1);
}

void handleAlarm(int ignore) {
    printf("0.5 sec passed..\n");
}
```

Προγραμματισμός συστημάτων UNIX/POSIX

Φετινό θέμα:

*Προγραμματιστικές τεχνικές που στοχεύουν
σε επιδόσεις*



MYY502

Βελτιστοποιήσεις με στόχο τις επιδόσεις

- ❖ Σε αρκετές περιπτώσεις δεν αρκεί να φτιάξουμε ένα πρόγραμμα που δουλεύει μόνο **σωστά** αλλά θέλουμε να δουλεύει και **όσο το δυνατόν πιο αποδοτικά / γρήγορα**.
- ❖ Για να γίνει αυτό απαιτούνται πολλές και διαφορετικές βελτιστοποιήσεις (optimization) του προγράμματός μας ώστε ο κώδικας μας να είναι αποδοτικότερος
- ❖ Σε πολλές περιπτώσεις, ο μεταφραστής είναι σε θέση να κάνει τις βελτιστοποιήσεις αυτές **αυτόματα**, μόνος του, χωρίς να αλλάξουμε τίποτε στο πρόγραμμά μας – αρκεί να του το πούμε.
 - Οι σύγχρονοι μεταφραστές είναι πάρα πολύ ικανοί σε αυτή τη δουλειά.



❖ Π.χ. ο GCC:

- Κατά τη μετάφραση ενός προγράμματος μπορούμε να του δώσουμε ορίσματα τύπου “-O”: -01, -02, -03 (-00 είναι το default)
 - Σε πολλές περιπτώσεις έχουμε εντυπωσιακά αποτελέσματα (παρότι αργεί λίγο παραπάνω για τη μετάφραση μεγάλων προγραμμάτων)
 - Να θυμόμαστε ότι σε μερικές περιπτώσεις οι αλλαγές που γίνονται στον κώδικα είναι πολύ δραστικές και ίσως σε ακραίες περιπτώσεις οι βελτιστοποιήσεις να αλλάζουν λίγο τη συμπεριφορά του προγράμματος (και να δυσκολεύεται το debugging) – θέλει προσοχή.
 - Τι σημαίνει το κάθε όρισμα (πόσο μάλλον **πώς** το κάνει αυτό που κάνει), δεν μας απασχολεί σε αυτό το μάθημα
-
- ❖ Εκτός από αυτό (ή μάλλον, σε συνδυασμό με αυτό), ενδιαφερόμαστε και για προγραμματιστικές τεχνικές μιας και καλύπτουν και περιπτώσεις που δεν μπορεί κανένας μεταφραστής να πιάσει.

Μερικές στοιχειώδεις βελτιστοποιήσεις |

❖ Αποφυγή περιττών υπολογισμών μέσα σε loops

```
for (i = 0; i < N; i++) {  
    s[i] = (x+3*y*z)*s[i];  
}
```

```
a = x+3*y*z;  
for (i = 0; i < N; i++) {  
    s[i] = a*s[i];  
}
```

- ❖ Οι μεταφραστές από μόνοι τους, ανακαλύπτουν αρκετές εκφράσεις (όχι όλες) που δεν εξαρτώνται από την επανάληψη του βρόχου (*loop-invariant*) και κάνουν το παραπάνω αυτόματα

➤ Καλό είναι να μην τους εμπιστευόμαστε 100%

Μερικές στοιχειώδεις βελτιστοποιήσεις II

❖ Μείωση περιττών κλήσεων σε συναρτήσεις

```
void capitalize(char *s) {  
    int i;  
  
    for (i = 0; i < strlen(s); i++)  
        s[i] = toupper(s[i]);  
}
```

```
void capitalize(char *s) {  
    int i, len = strlen(s);  
  
    for (i = 0; i < len; i++)  
        s[i] = toupper(s[i]);  
}
```

❖ Για μήκος 1 MiB

➤ 25.5 sec

(Στο desktop μου)

❖ Για μήκος 1 MiB

➤ 0.01 sec

Μεταβλητές “register” I

- ❖ Μία μεταβλητή στην C μπορεί να δηλωθεί ως “register”, π.χ.

```
register int x;  
register float y;
```

- ❖ Ο χαρακτηρισμός “register” δηλώνει στον μεταφραστή ότι η μεταβλητή αυτή χρησιμοποιείται πολύ συχνά και καλό είναι να μην την τοποθετήσει κάπου στην μνήμη (αργή) αλλά σε έναν καταχωρητή μέσα στην CPU.
- ❖ Δύο σημαντικές παρατηρήσεις:
 1. Ο μεταφραστής δεν είναι υποχρεωμένος να το σεβαστεί (δηλαδή μπορεί να το αγνοήσει)
 2. Δεν μπορούμε να πάρουμε τη διεύθυνση μίας μεταβλητής register (το &x δεν δουλεύει)



Μεταβλητές “register” II

- ❖ Οι σύγχρονοι μεταφραστές κάνουν εξαιρετική διαχείριση των καταχωρητών της CPU μετά από ανάλυση του προγράμματός μας και αποφασίζουν μόνοι τους πώς και που θα αποθηκευτούν οι διάφορες μεταβλητές. Έτσι
 - Σχεδόν πάντα κοιτούν να αγνοήσουν το “register”
 - Αν δεν το αγνοήσουν, κάποιες φορές οδηγούμαστε σε χειρότερες επιδόσεις από αυτές που θα είχαμε αν έπαιρνε όλες τις αποφάσεις ο μεταφραστής μόνος του
 - Επομένως, καλό είναι να μην τις πολυχρησιμοποιούμε σε προγράμματα γενικού σκοπού παρά μόνο σε εξειδικευμένες περιπτώσεις και σε περιπτώσεις που ο μεταφραστής δεν έχει την ικανότητα να κάνει πολλές βελτιστοποιήσεις (π.χ. σε ενσωματωμένα συστήματα).



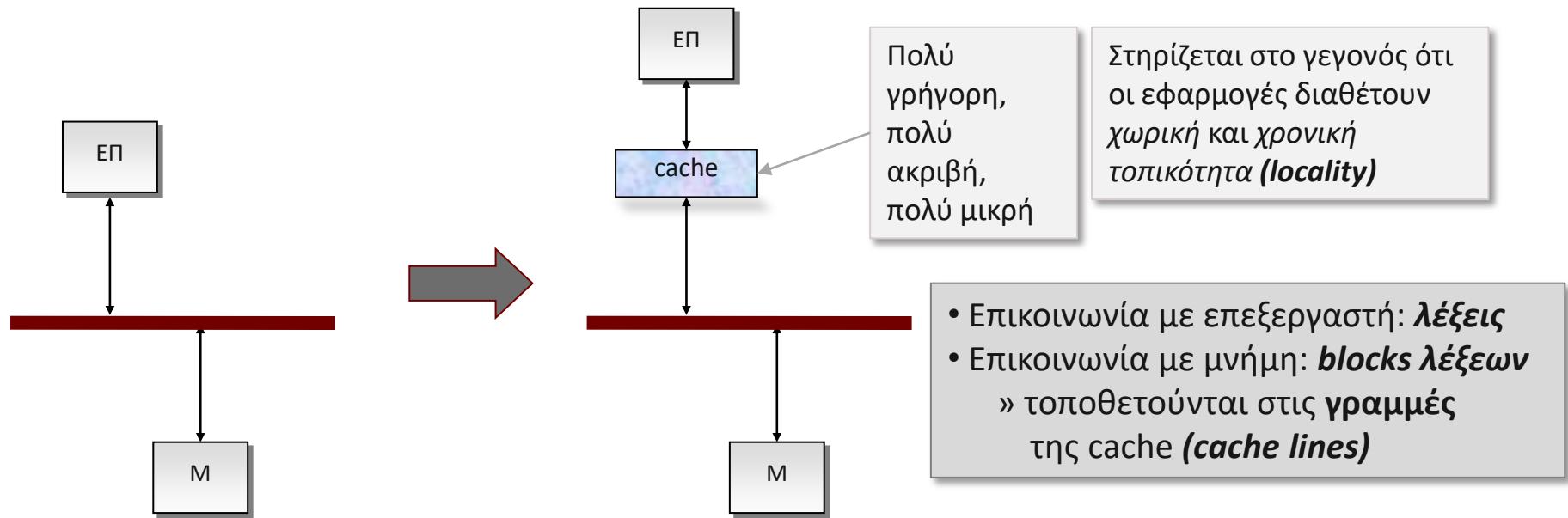
Προγραμματίζοντας με γνώση της ιεραρχίας μνήμης

- ❖ Οι εντολές και τα δεδομένα ενός προγράμματος βρίσκονται αποθηκευμένα στην μνήμη, όπως γνωρίζουμε.
- ❖ Όμως, εδώ και πολλά χρόνια, η μνήμη του υπολογιστή δεν αποτελείται μόνο από ένα «κομμάτι».
- ❖ Υπάρχει ολόκληρη ιεραρχία από διαφορετικές μνήμες, παρότι ο επεξεργαστής θεωρεί ότι μιλάει μόνο με την «κύρια μνήμη»:
 - Η βασική (κύρια) μνήμη, γνωστή δυστυχώς και ως «η RAM του συστήματος»
 - Η κρυφή μνήμη πρώτου επιπέδου (L1 cache)
 - Η κρυφή μνήμη δεύτερου επιπέδου (L2 cache)
 - Η κρυφή μνήμη τρίτου επιπέδου (L3 cache)
- ❖ Όλες αυτές αποτελούν μαζί τη «μνήμη» του συστήματος όπου αποθηκεύονται εντολές και δεδομένα.
 - Όταν μία μεταβλητή αποθηκεύεται στη μνήμη, που ακριβώς πάει;
 - Μπορώ να εκμεταλλευτώ κάπως την ιεραρχία αυτή για αυξημένες επιδόσεις;



Σύνοψη της ιεραρχίας μνήμης

- ❖ Επεξεργαστής: ταχύτατος
- ❖ Μνήμη: αργή (και μάλιστα η διαφορά ταχύτητας αυξάνεται)
- ❖ Βασική λύση: *κρυφή μνήμη (cache)*

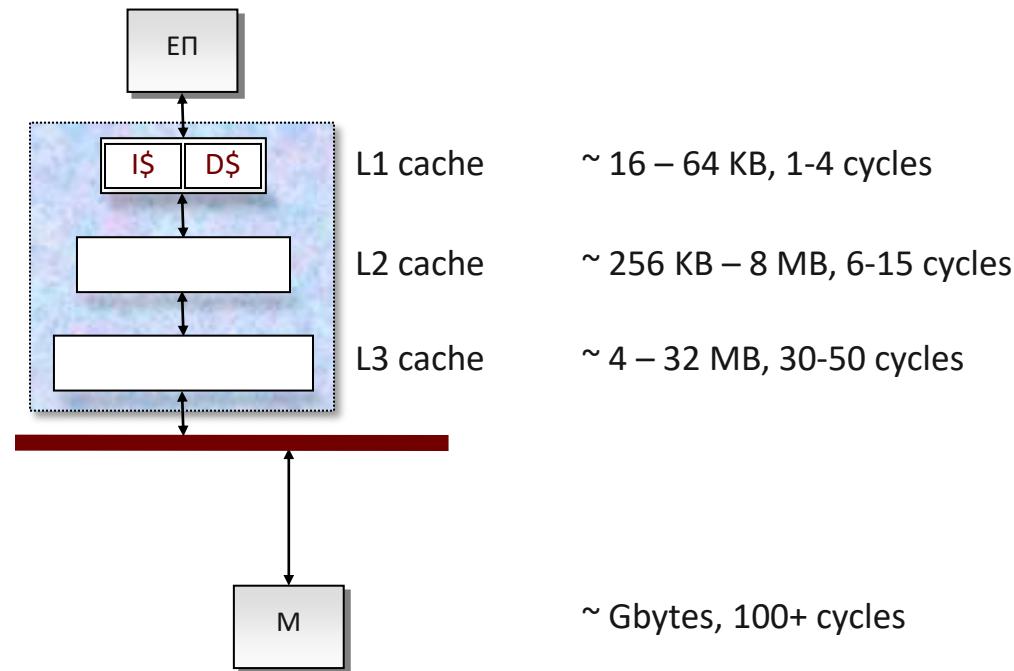


Συμπεριφορά σε αναγνώσεις (read)

- ❖ Read **hit** : η λέξη υπάρχει ήδη σε μία γραμμή της cache
 - Η ζητούμενη λέξη ξεχωρίζει από την γραμμή της cache και παραδίδεται στον επεξεργαστή.
- ❖ Read **miss** : η λέξη δεν υπάρχει σε καμία γραμμή της cache
 - Το μπλοκ που περιλαμβάνει τη ζητούμενη λέξη προσκομίζεται από τη μνήμη, τοποθετείται σε μία γραμμή της cache και η λέξη παραδίδεται στον επεξεργαστή
 - **Μεγάλη καθυστέρηση**
- ❖ Hit ratio = # προσπελάσεων που ήταν hit / συνολικό # προσπελάσεων
 - Ποσοστά 70 – 95% πολύ συχνά, ακόμα και με μικρές cache

Γενίκευση σε πολλαπλά επίπεδα (ιεραρχία caches)

- ❖ Για μείωση του κόστους αστοχίας (miss)
- ❖ 2-3 επίπεδα από αυξανόμενου μεγέθους και μειούμενου κόστους / ταχύτητας caches:



- ❖ Απαραίτητη μιας και εξοικονομεί πολύ χρόνο
 - Πολύ γρηγορότερη από την κύρια μνήμη
 - Πολύ μικρότερη από την κύρια μνήμη
- ❖ Ότι δεδομένο χρησιμοποιήσει η διεργασία μας (δηλαδή ο επεξεργαστής),
 - Το φέρνει και το φυλάει
 - Φέρνει και τα γειτονικά του δεδομένα (φέρνει ένα ολόκληρο μπλοκ μνήμης)
- ❖ Εφόσον το πρόγραμμα έχει τοπικότητα (δηλαδή χρησιμοποιεί σχετικά λίγα και γειτονικά δεδομένα για αρκετή ώρα), έχουμε πολύ μεγάλη βελτίωση στην ταχύτητα προσπέλασης των δεδομένων.

Πώς να καταστρέψετε την cache

❖ Τι γίνεται εδώ;

```
for (i = 0; i < 10000; i++)  
    for (j = 0; j < 10000; j++)  
        a[i][j] = 2*b[i][j];
```

```
for (j = 0; j < 10000; j++)  
    for (i = 0; i < 10000; i++)  
        a[i][j] = 2*b[i][j];
```

Χρόνος εκτέλεσης στο (παλιό) φορητό μου:

1.2 sec

Χρόνος εκτέλεσης στο (παλιό) φορητό μου:

14 sec

Αυξημένη τοπικότητα (προσπέλαση γειτονικών στοιχείων το ένα μετά το άλλο λόγω της προσπέλασης κατά γραμμές)

Κακή τοπικότητα λόγω της προσπέλασης κατά στήλες. Φέρνοντας το b[0][0], έρχεται ολόκληρη γραμμή με τα b[0][1], b[0][2], ..., b[0][7] (και ίσως και παραπάνω). Όμως στην επόμενη επανάληψη του i, θέλουμε το b[1][0]! Επομένως έφερε τη γραμμή άσκοπα, πρέπει να φέρει νέα γραμμή ... η οποία επίσης είναι άχρηστη για τη συνέχεια κ.ο.κ.

Οδηγίες για προγράμματα cache-friendly

- ❖ Στοχεύστε κυρίως στα loops που τρώνε τον περισσότερο χρόνο εκτέλεσης
- ❖ Αυξήστε την τοπικότητα σιγουρεύοντας ότι στο πιο συχνά χρησιμοποιούμενο εσωτερικό βρόχο δεν δημιουργούνται πολλά cache misses.
- ❖ Δείτε και τις επόμενες διαφάνειες ☺



Padding I

- ❖ Οι σύγχρονοι επεξεργαστές έχουν cache με γραμμές 32, 64 ή 128 bytes. Πρέπει να γνωρίζουμε το μέγεθος όταν βελτιστοποιούμε τις επιδόσεις.
- ❖ Για παράδειγμα, ας θεωρήσουμε ότι έχουμε cache με γραμμή μεγέθους **LINESIZE = 32 bytes** και θέλουμε να δημιουργήσουμε έναν πίνακα από τις παρακάτω δομές:

```
struct data {  
    int    a,b,c,d;    /* 4 bytes each */  
    float z,y;        /* 4 bytes each */  
}
```

- ❖ Αν και οι μεταφραστές μπορεί να κάνουν κάποιες τροποποιήσεις στον χώρο της δομής, λογικά το μέγεθος του struct είναι 24 bytes. Αυτό σημαίνει ότι πιάνει $\frac{3}{4}$ μίας γραμμής cache.
- ❖ Έτσι, π.χ. ενώ το στοιχείο 0 ενός πίνακα από τέτοια struct πέφτει σε μία γραμμή, το στοιχείο 1 πέφτει σε ΔΥΟ γραμμές (8 bytes από την μία και 16 bytes από την επόμενη)
 - Άρα η προσπέλαση του στοιχείου 1 απαιτεί προσκόμιση ΔΥΟ γραμμών, ενώ το 0 θέλει μόνο μία!!

Padding II

- ❖ Λύση: **padding** («γέμισμα») με άχρηστα bytes ώστε να πιάνω χώρο που είναι ακέραιο πολλαπλάσιο του LINESIZE:

```
struct data {  
    int    a,b,c,d;  
    float  x,y;  
    char   pad[8]; /* Άχρηστο, για να φτάσω 32 bytes */  
}
```

- ❖ Κάθε στοιχείο ενός πίνακα από τέτοια structs πιάνει χώρο ακριβώς όσο και 1 γραμμή της cache.
 - Επομένως αρκεί να προσκομιστεί 1 μόνο γραμμή για κάθε στοιχείο!
- ❖ «Πληρωμή» σε χώρο για να κερδίσουμε ταχύτητα

Alignment

- ❖ Τα μπλοκ από την κύρια μνήμη πάνε σε γραμμές τις cache.
- ❖ OK, το padding, αλλά πώς εξασφαλίζω ότι το `malloc()` που θα κάνω για τον πίνακα με τα `structs` θα ξεκινάει ΑΚΡΙΒΩΣ στην αρχή ενός μπλοκ (δηλαδή θα ξεκινά από ακέραιο πολλαπλάσιο του `LINESIZE`)?
 - ❖ Π.χ. αν το `malloc()` επιστρέψει τη θέση 32010?
 - Τότε κανένα στοιχείο του πίνακα δεν θα πέφτει στην αρχή μίας γραμμής!!
 - ❖ Λύση: **alignment** (στοίχιση) διευθύνσεων σε ακέραια πολλαπλάσια του `LINESIZE`.
 - Είτε αυτόματα με ειδικές εντολές του μεταφραστή (μη αποδεκτή/μεταφέρσιμη C)
 - Είτε με το χέρι, παίρνοντας λίγο παραπάνω χώρο



«Χειροποίητο» alignment

- ❖ Έστω ότι θέλω να δεσμεύσω χώρο S bytes. Πώς μπορώ να ξέρω αν η `malloc()` θα μου επιστρέψει διεύθυνση η οποία είναι πολλαπλάσιο του LINESIZE?
- ❖ Επειδή δεν μπορώ να το γνωρίζω αυτό, η ιδέα είναι η εξής:
 - a) Ζητάω λίγο παραπάνω χώρο, συγκεκριμένα ($LINESIZE - 1$) παραπάνω bytes.
 - b) Από αυτά που μου δόθηκαν χρησιμοποιώ τα S bytes ακριβώς, ξεκινώντας όμως από τη θέση που σίγουρα είναι πολλαπλάσιο του LINESIZE
- Δέσμευση παραπάνω μνήμης:

```
whole = malloc(S + LINESIZE - 1);
```
- Υποθέτοντας (μιας και συμβαίνει πάντα στην πράξη) ότι το LINESIZE είναι μία δύναμη του 2 (32, 64, 128 bytes), η διεύθυνση που είναι πολλαπλάσιο του LINESIZE είναι η:

```
alignedstart = (whole + LINESIZE - 1) & ~(LINESIZE - 1);
```

και αυτή πρέπει να χρησιμοποιήσω ως αρχή των S bytes.

(Γιατί;;;)

Βοήθεια: αν το LINESIZE είναι 2^k , τα πολλαπλάσιά του έχουν πάντα μηδενικά στα πρώτα/δεξιότερα k bits τους

Ερώτηση

- ❖ Τι συμπεριφορά θα έχει το παρακάτω πρόγραμμα;
 - Τι γίνεται στην εκτέλεση της `sumarray()` όταν αυξάνει το `stride`; Όταν αυξάνει το `size`;
 - Σε τι θα χρησίμευε; Μπορώ να καταλάβω κάτι από τους χρόνους;

```
#define MAX (1 << 22)          /* Up to 4 Mbytes */
int array[MAX];

for (size = MAX; size >= 0; size--) {
    for (stride = 1; stride <= MAXSTRIDE; stride++) {
        sumarray(size, stride);
    }
}

void sumarray(int size, int stride) {
    int i, sum = 0;
    for (i = 0; i < size; i += stride)
        sum += array[i];
}
```