

Διάλεξη 0 - Καλημέρα Κόσμε

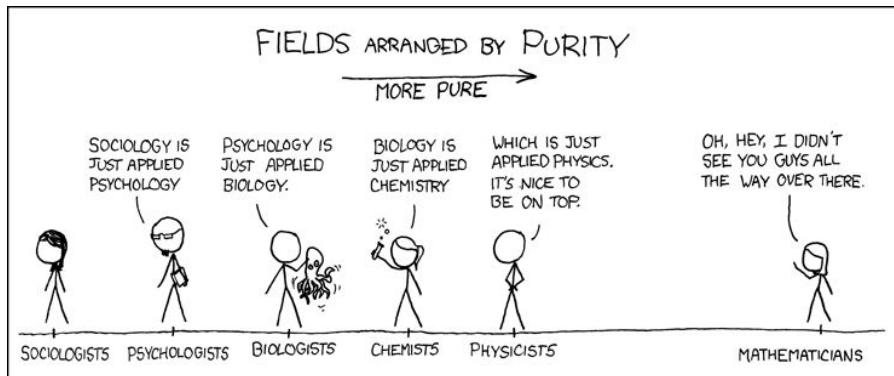
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

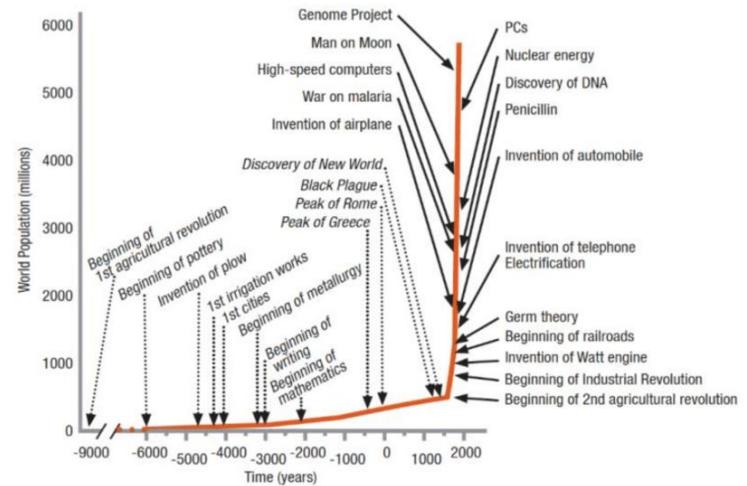
Ενδεικτικοί Λόγοι

- Ευελιξία (θεματική και γεωγραφική)



- Πνευματικά απαιτητική δουλειά

- Στην αιχμή των εξελίξεων



- Cool

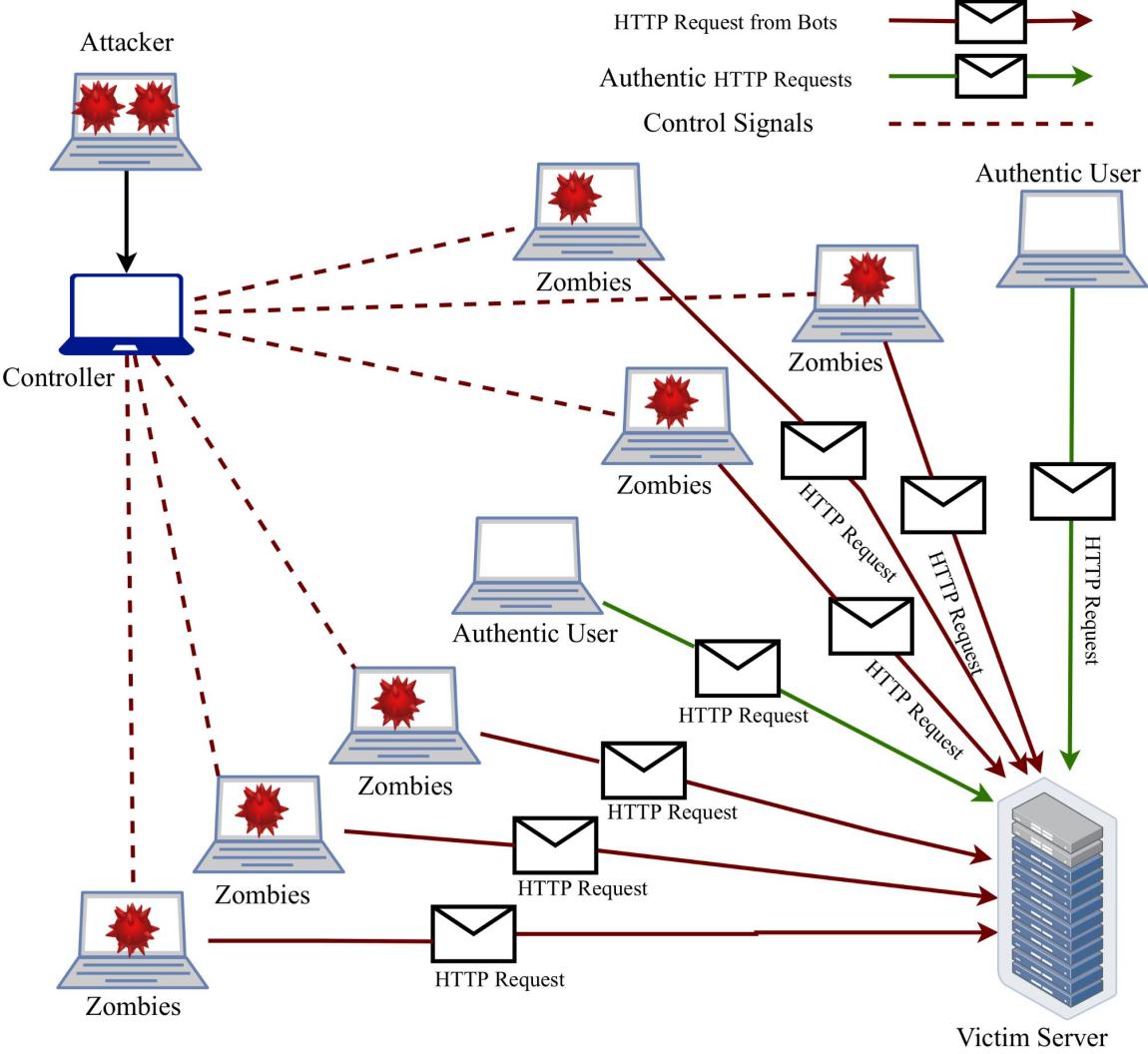
Διαδικαστικά 3/5 - Βαθμολογία

- Αν είστε πρωτοετείς:
 - $50\% * \text{Τελική Εξέταση} + 30\% * \text{Ασκήσεις} + 20\% * \text{Εργαστήριο}$
 - Άλλιώς:
 - $70\% * \text{Τελική Εξέταση} + 30\% * \text{Ασκήσεις}$
-
1. **Τελική Εξέταση:** γραπτή με κλειστά βιβλία
 2. **Ασκήσεις:** προαιρετικές αλλά η επίλυσή τους βοηθάει σημαντικά
 3. **Εργαστήριο:** υποχρεωτική παρουσία για τους πρωτοετείς (μέχρι 2 απουσίες)

Browsing Time

progintro, sign ups, φόρμα, σημειώσεις, συγγράμματα, delos

Denial of Service Attacks



Τι είναι ο Υπολογιστής;

Μια κατασκευή που έχει την ικανότητα να επεξεργάζεται ένα σύνολο από δεδομένα που του δίνονται και να παράγει τα απαιτούμενα αποτελέσματα.

Γιατί χρησιμοποιούμε υπολογιστές;

1. Ταχύτητα στην επεξεργασία δεδομένων
2. Μνήμη διαθέσιμη για αποθήκευση δεδομένων

Τι είναι ο Προγραμματισμός;

Προγραμματισμός είναι ο σαφής καθορισμός μίας διαδικασίας, σαν ένα σύνολο από εντολές (το πρόγραμμα), που περιγράφει λεπτομερώς τα βήματα που πρέπει να γίνουν για να επιλυθεί ένα πρόβλημα υπολογισμού.

Poll: Έχει κάποιο παιδί προγραμματίσει;

Poll: Έχει κάποιο παιδί προγραμματίσει;

Κάποια παραδείγματα

προγραμμάτων που ξέρετε;

Poll: Έχει κάποιο παιδί προγραμματίσει;

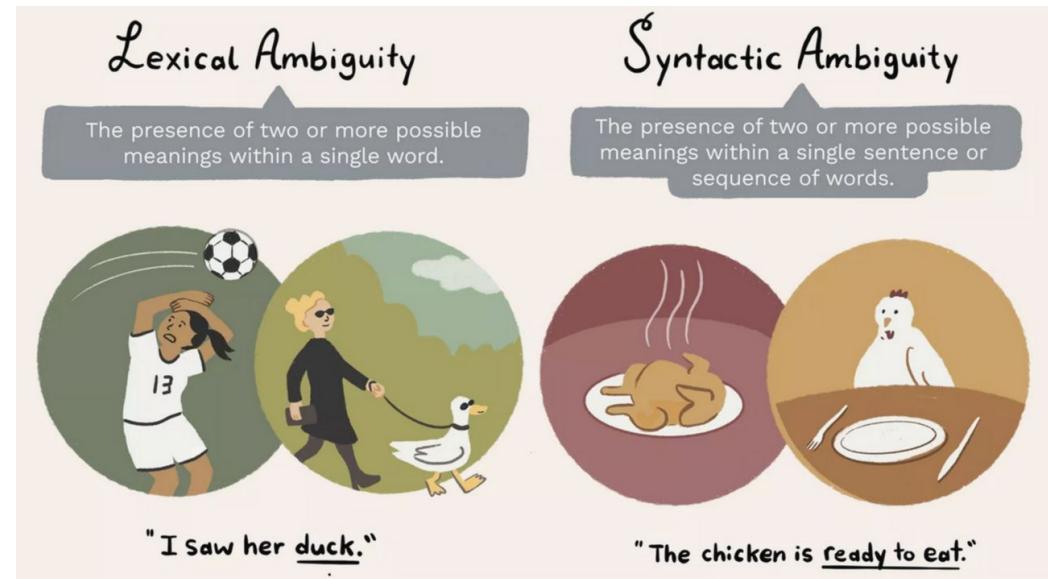
Κάποια παραδείγματα
προγραμμάτων που ξέρετε;

Πρόγραμμα είναι η καταγραφή της
επίλυσης ενός προβλήματος



Γιατί επινοήσαμε τις Γλώσσες Προγραμματισμού;

- > Joe says to his programmer friend Charlie:
- > "Go to the store and buy a loaf of bread. If they have eggs, buy a dozen."
- > Charlie returns with 12 loaves of bread.



Τι είναι η Γλώσσα Προγραμματισμού;

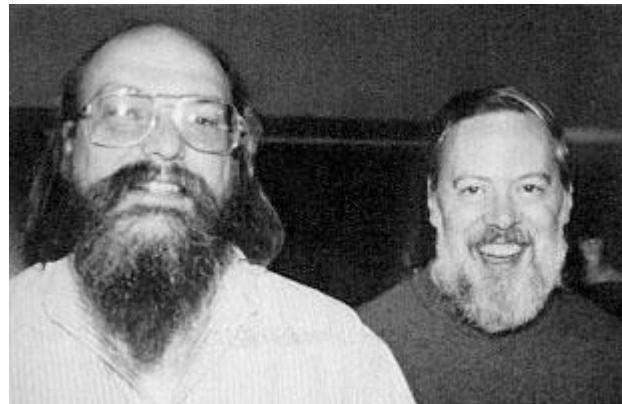
Μια γλώσσα που μας επιτρέπει να επικοινωνούμε εντολές στον υπολογιστή. Μια καλή γλώσσα προγραμματισμού δεν επιτρέπει αμφισημία.

Bonus: Η γλώσσα προγραμματισμό σε συνδυασμό με τον υπολογιστή μπορεί να χρησιμοποιηθεί για να επιλύσει αμφισημίες ανάμεσα σε προγραμματιστές.

Γιατί η Γλώσσα Προγραμματισμού C;

- Απαιραίτητη στο πρόγραμμα σπουδών
- Δημοφιλής
- Χρησιμοποιείται παντού και (σχεδόν) για τα πάντα

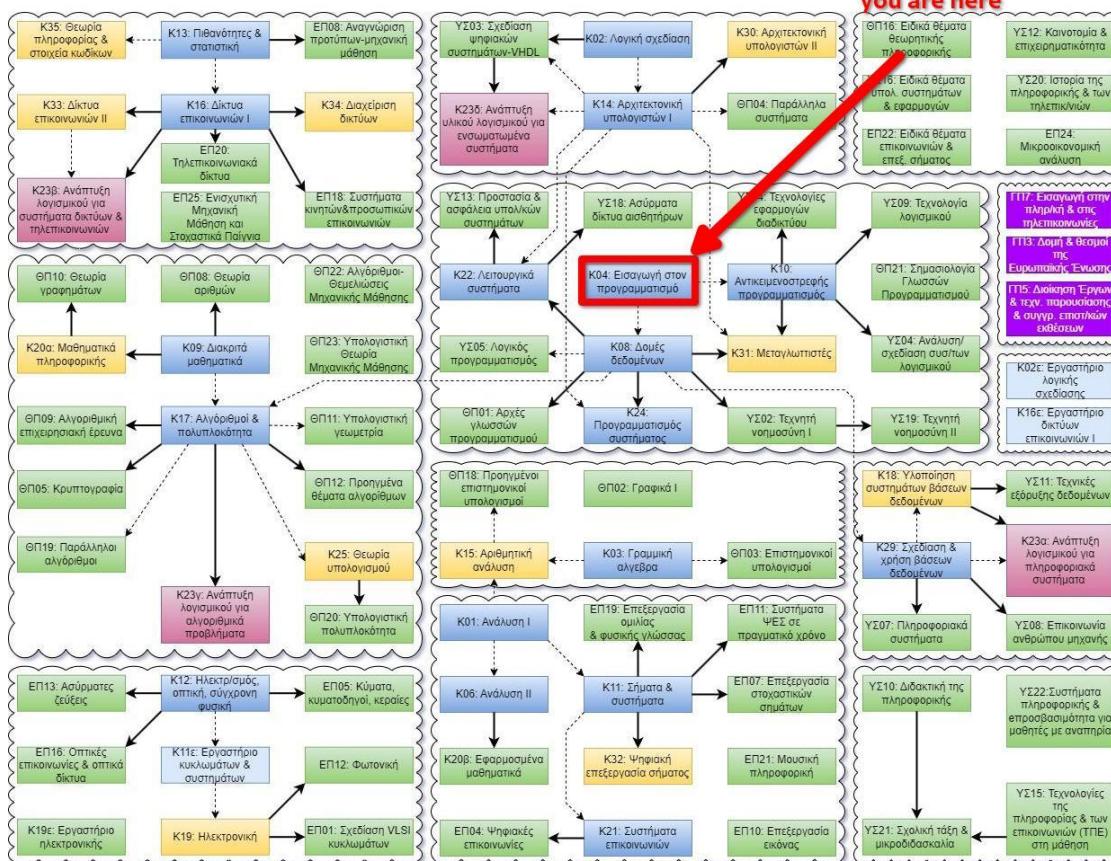
Ken Thompson
(co-creator
UNIX and B)



Dennis Ritchie
(creator of C,
co-creator of
UNIX and B)

~1970

you are here



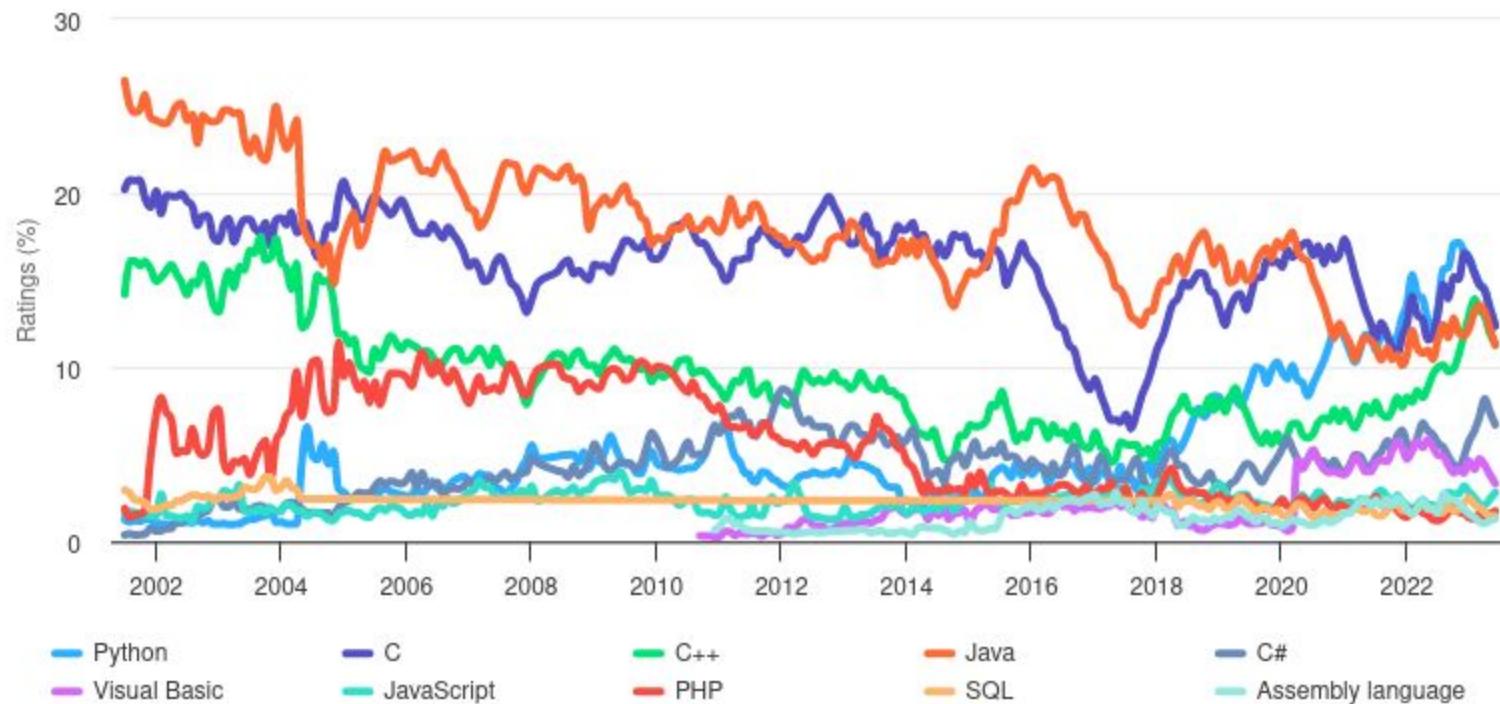
ΥΠΟΜΝΗΜΑ

K01: Ανάλυση I	Μαθήματα κορμού	Υ10: Διδακτική της πληροφορικής	Γνωστικό πεδίο
Ι11: Εισαγωγή στην πληρική & στις τηλεπικοινωνίες	Μαθήματα γενικής παιδείας	K25: Θεωρία υπολογισμού	A -----> B Το A αποτελεί συνιστώμενο προστατούμενο του B
K11e: Εργαστήριο κυκλωμάτων & συστημάτων	Μαθήματα εργαστηρίου	Kat επιλογή υποχρεωτικά μαθήματα(ΕΥΜ)	A -----> B Το A αποτελεί συνιστώμενο προστατούμενο του B

Στις πιο δημοφιλείς Γλώσσες από το 1970 [\[tiobe.com\]](http://tiobe.com)

TIOBE Programming Community Index

Source: www.tiobe.com



Γλώσσα C: Το "αγωνιστικό" των Γλωσσών

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) JRuby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) JRuby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) JRuby	19.84

[Programming](#)
[Language Shootout](#)

Το Πρόγραμμα Hello World

```
/* File: helloworld.c */

#include <stdio.h>

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

Ας το τρέξουμε με έναν [online compiler](#)



Για την επόμενη φορά

- Από τις σημειώσεις του κ. Σταματόπουλου μέχρι την σελίδα 19.
- Ολοκληρώστε τις εγγραφές στα εργαλεία του μαθήματος.
- Γραφτείτε σε κάποιο εργαστήριο!
- Αν θέλετε φέρτε υπολογιστή μαζί σας.

Διάλεξη 1 - Η Γραμμή Εντολών

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός



Joshua Gross · Follow

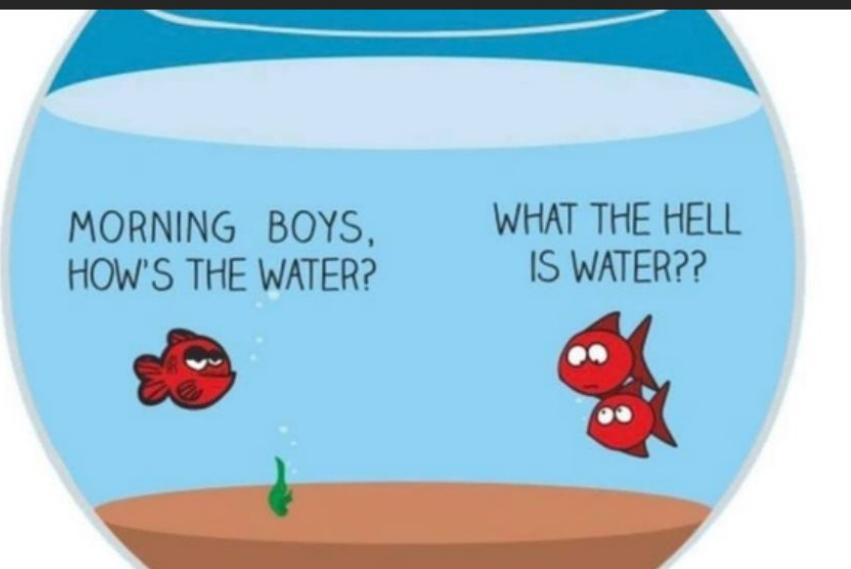
Associate Professor of Computer Science at CSUMB · Updated Sat

X

Why do universities still teach C and C++?

There's an old joke. Two fish are swimming along, and another, older fish swims by and says, "Water's nice today, huh?"

A moment later, one of the two fish turns to the other and say, "What's water?"



I'm writing this answer on a Mac. The operating system was written in C.

I'm writing this on Firefox. Most of the core is (AFAIK) written in C++, along with (probably) some Objective-C for the UI layer.

The wireless keyboard and mouse I'm using may have device drivers written in C, and they probably are also themselves programmed in C.

Though some of the specifics might change, the vast majority of software in use today was written in C, C++, or a derivative language. Sure, there's some COBOL, but less each year. And Python is used extensively, but often not for what we would normally call software. And it's not entirely clear how much C heritage is carried in JavaScript, but even so, C and its derivative languages are our water.

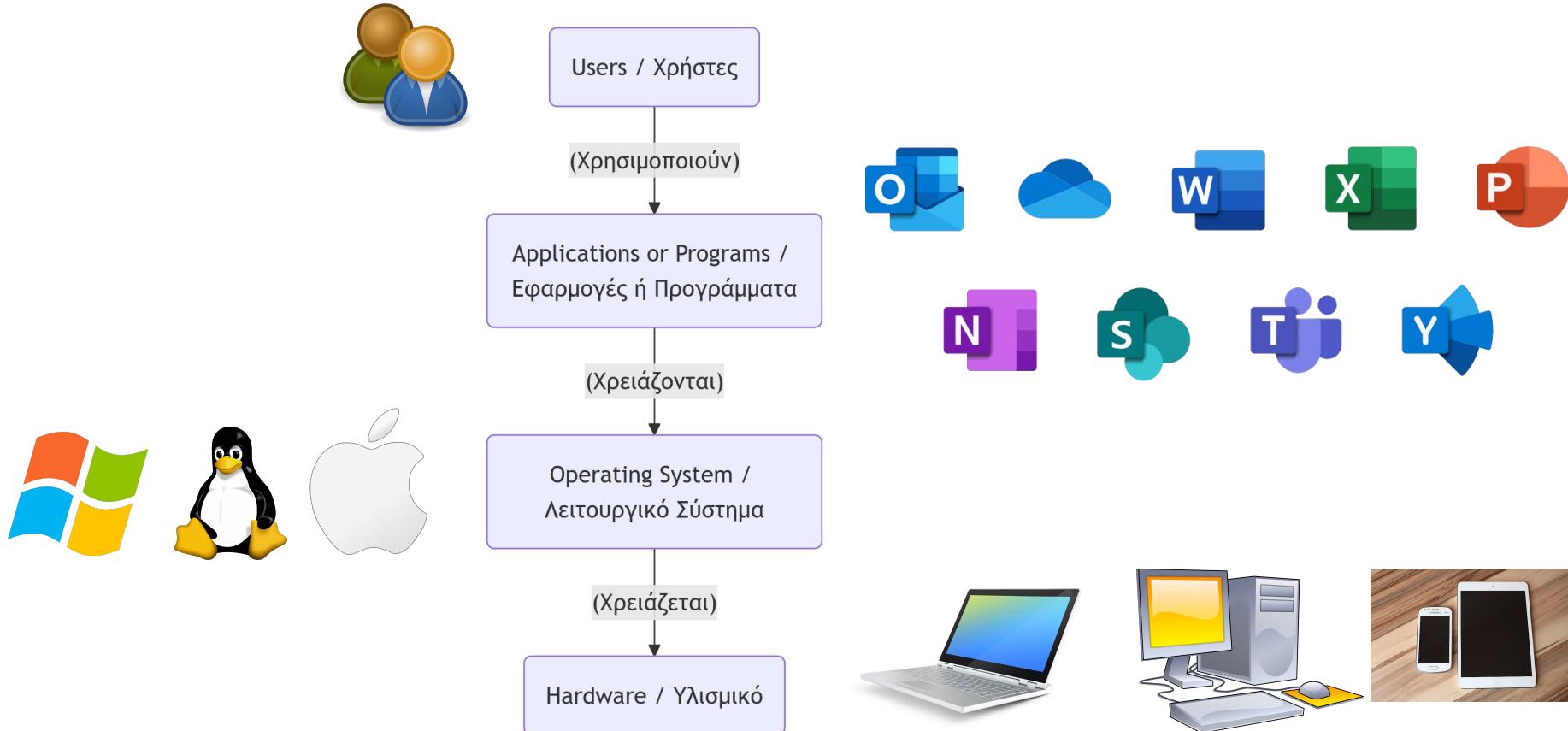
And fish should know what water is.

406.8K views · View 2,608 upvotes · View 24 shares · Answer requested by
Miguel Paraz

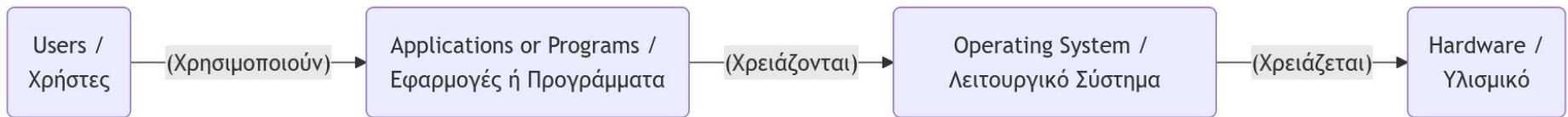
1 of 13 answers

<https://qr.ae/p2wHA3>

Βασική Δομή Υπολογιστικών Συστημάτων Σήμερα

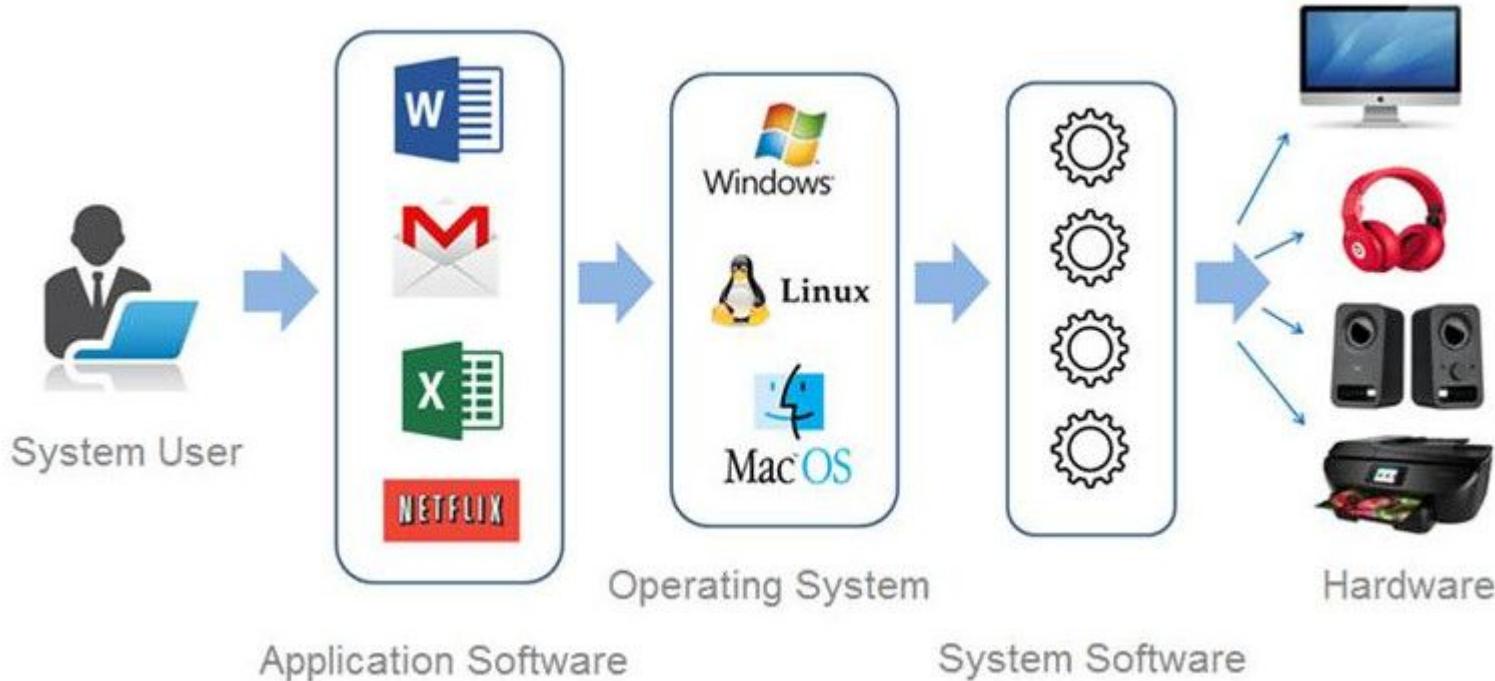


Αρχιτεκτονική Υπολογιστικών Συστημάτων



Παραδείγματα;

Αρχιτεκτονική Υπολογιστικών Συστημάτων



Operating Systems (Λειτουργικά Συστήματα)

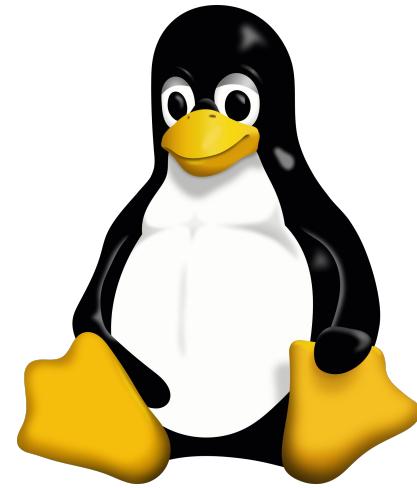
Operating System: Το λογισμικό του υπολογιστή που είναι υπεύθυνο για:

- Την διαχείριση του υλικού του υπολογιστή (συσκευές)
- Την διαχείριση πόρων του συστήματος (πόση μνήμη μπορεί να χρησιμοποιηθεί από ένα πρόγραμμα, πότε θα πάρει κύκλους στον επεξεργαστή, κ.ο.κ.)
- Προσφορά υπηρεσιών σε προγράμματα χρηστών (αποθήκευση πληροφοριών στον δίσκο, μεταφορά πληροφοριών μέσω δικτύου, κ.ο.κ.)

Γιατί να μην είναι το πρόγραμμά μας υπεύθυνο για αυτά;

Linux

- Open-source operating system βασισμένο στο [Unix](#) OS.
- Βασισμένο στο Linux kernel (πυρήνα), που ξεκίνησε από τον Linus Torvalds το 1991.
- Γραμμένο κυρίως σε [γλώσσα C](#)
- Διαδεδομένο σε χρήση
 - Στηρίζει το 90% των web servers
 - Αποτελεί την βάση του Android - τρέχει στην πλειοψηφία των κινητών
 - Είναι η αποκλειστική επιλογή για supercomputers (υπερυπολογιστές)



Δύο Τρόποι Διάδρασης με Λειτουργικά Συστήματα

- Graphical User Interface (GUI)
Γραφική Διεπαφή Χρήστη :'
- Command Line Interface (CLI)
Διεπαφή Γραμμής Εντολών, Τερματικό (Terminal), Κονσόλα (Console), Κέλυφος (Shell) - Διαφορές



```
mark@linux-desktop: /tmp/tutorial
File Edit View Search Terminal Help
Setting up tree (1.7.0-5) ...
Processing triggers for man-db (2.8.3-2) ...
mark@linux-desktop:/tmp/tutorial$ tree
.
├── another
├── combined.txt
├── dir1
├── dir2
│   ├── dir3
│   ├── test_1.txt
│   ├── test_2.txt
│   └── test_3.txt
└── dir4
    └── dir5
        └── dir6
    └── folder
    └── output.txt

8 directories, 5 files
mark@linux-desktop:/tmp/tutorial$
```



Στο Μάθημα θα επιμείνουμε στην χρήση CLI



Τρέχοντας Προγράμματα (Εντολές) σε Linux Shell

Για να τρέξουμε ένα πρόγραμμα με N ορίσματα πρέπει να γράψουμε:

Πρόγραμμα Όρισμα1 Όρισμα2 ... Όρισμα N

Όρισμα (Argument) / Δεδομένο Εισόδου (Input Data): μια τιμή που μπορεί να χρησιμοποιηθεί από το πρόγραμμα κατά το τρέξιμό του.

Δεδομένο Εξόδου (Output): η τιμή που επιστρέφει το πρόγραμμα όταν τελειώσει τον υπολογισμό του.

Τρέχοντας Προγράμματα (Εντολές) σε Linux Shell

Για να τρέξουμε ένα πρόγραμμα με N ορίσματα πρέπει να γράψουμε:

Πρόγραμμα Όρισμα1 Όρισμα2 ... Όρισμα N

Παράδειγμα:

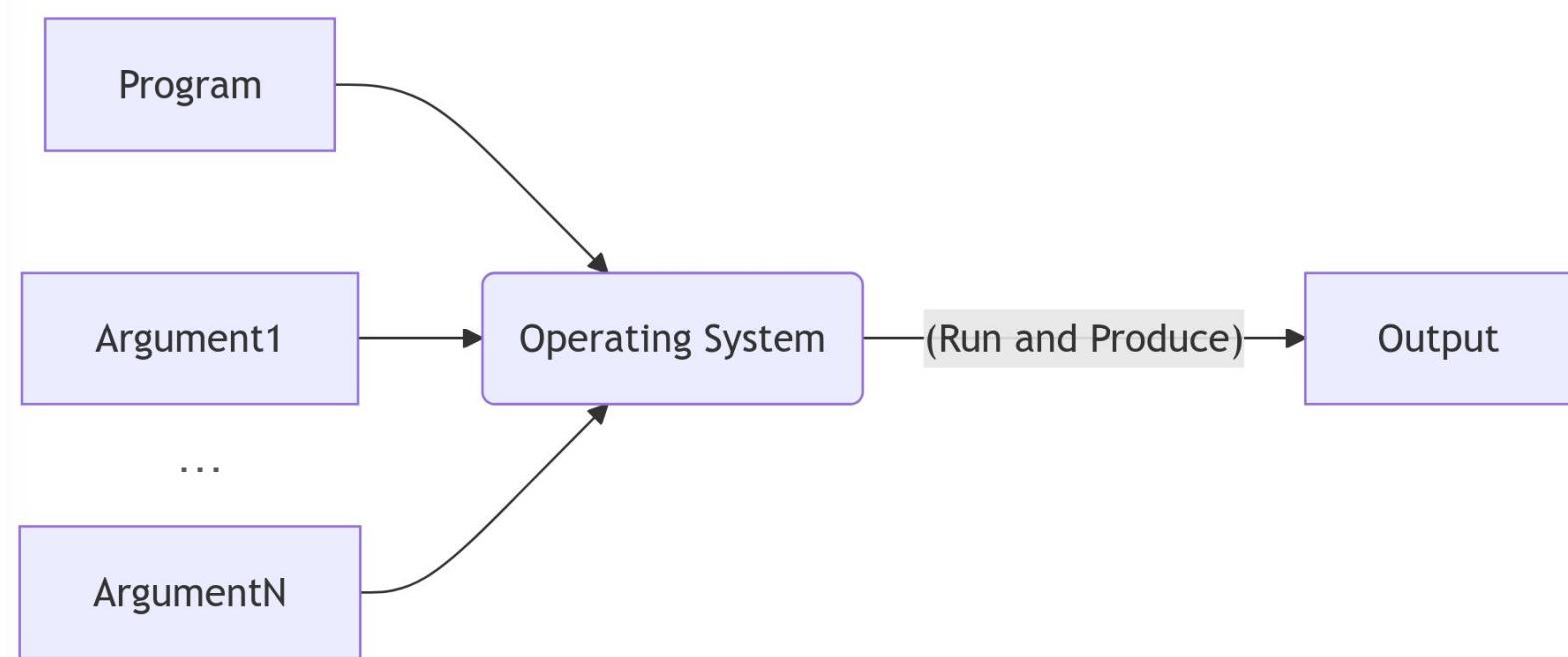
/bin/echo hello good world

Ποιο είναι το πρόγραμμα; Πόσα ορίσματα έχει;

Ας τρέξουμε το πρόγραμμα `/bin/echo`

Τρέχοντας Προγράμματα (Εντολές) σε Linux Shell

To Shell μας επιτρέπει να τρέξουμε προγράμματα και να πάρουμε outputs:



4 Τρόποι Πρόσβασης *nix Γραμμής Εντολών

Ταξινομημένα από τον πιο πρακτικό προς τον λιγότερο πρακτικό τρόπο για προγραμματισμό

1. Εγκαθιστάς Linux (ή macOS) στον υπολογιστή σου
2. Εγκαθιστάς WSL 2 στο Windows μηχάνημά σου
3. Κάνεις ssh (μέσω putty ή κονσόλας) στα μηχανήματα του εργαστηρίου linuxXY.di.uoa.gr
4. Συνδέσου στο google cloud console που έχεις στον προσωπικό σου λογαριασμό

Στα 1, 2, 4 είσαι διαχειριστής (root) στο σύστημά σου. Στο 3 είσαι χρήστης.

Το "τυφλό" σύστημα / Touch Typing / Blind Typing

Μπορεί να κάνει την ζωή σου πιο εύκολη

- <https://blindtyping.com/>
- <https://www.typingclub.com/>
- <https://www.typingstudy.com/>

Και άλλα πολλά - αναζητήστε πηγές

Stories in the life of a DevOps engineer

Μόλις ήρθε ένα alert



ALARM: "Maximum-CPU-Utilization" in



- AWS Notifications <no-reply@sns.amazonaws.co
- To: [REDACTED]

You are receiving this email because your Amazon CloudWatch Alarm "Maximum-CPU-Utilization" was triggered at 20:02:50 UTC.

View this alarm in the AWS Management Console:

<https://us-west-2.console.aws.amazon.com/> [REDACTED]

Alarm Details:

- Name: Maximum-CPU-Utilization
- Description: A CloudWatch Alarm that triggers when Maximum CPU Utilization Exceeds
- State Change: INSUFFICIENT_DATA -> ALARM
- Reason for State Change: Threshold Crossed: 1 datapoint [8.175136252270871 (23/05/21 19:50:00)] crossed the threshold of 8.175136252270871

Εντυχώς υπάρχει το "τηλεχειριστήριο": ssh

Το SSH (Secure Shell) είναι ένα πρωτόκολλο δικτύου που επιτρέπει την ασφαλή απομακρυσμένη Σύνδεση και διαχείριση ενός υπολογιστή ή server μέσω κρυπτογραφημένης επικοινωνίας.

Παραδείγματα Εντολών

```
thanassis@linux14:~$ who
thanassis pts/0          Oct  5 14:36 (110.220.111.247)
thanassis@linux14:~$ last
...
thanassis@linux14:~$ whoami
thanassis
thanassis@linux14:~$ id
uid=2622(thanassis) gid=1001(dep) groups=1001(dep)
thanassis@linux14:~$ users
thanassis
thanassis@linux14:~$ factor 2394872891
2394872891: 3259 734849
thanassis@linux14:~$ hostname
linux14
thanassis@linux14:~$ sleep 3
thanassis@linux14:~$ uptime
 14:40:23 up 7 days,  5:49,  3 users,  load average: 0.00, 0.00, 0.00
thanassis@linux14:~$ uname -a
linux linux14 5.4.0-163-generic #180-Ubuntu SMP Tue Sep 5 13:21:23 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
# Interactive commands follow
thanassis@linux14:~$ top
thanassis@linux14:~$ man
thanassis@linux14:~$ ps
```

Filesystem (Σύστημα Αρχείων)

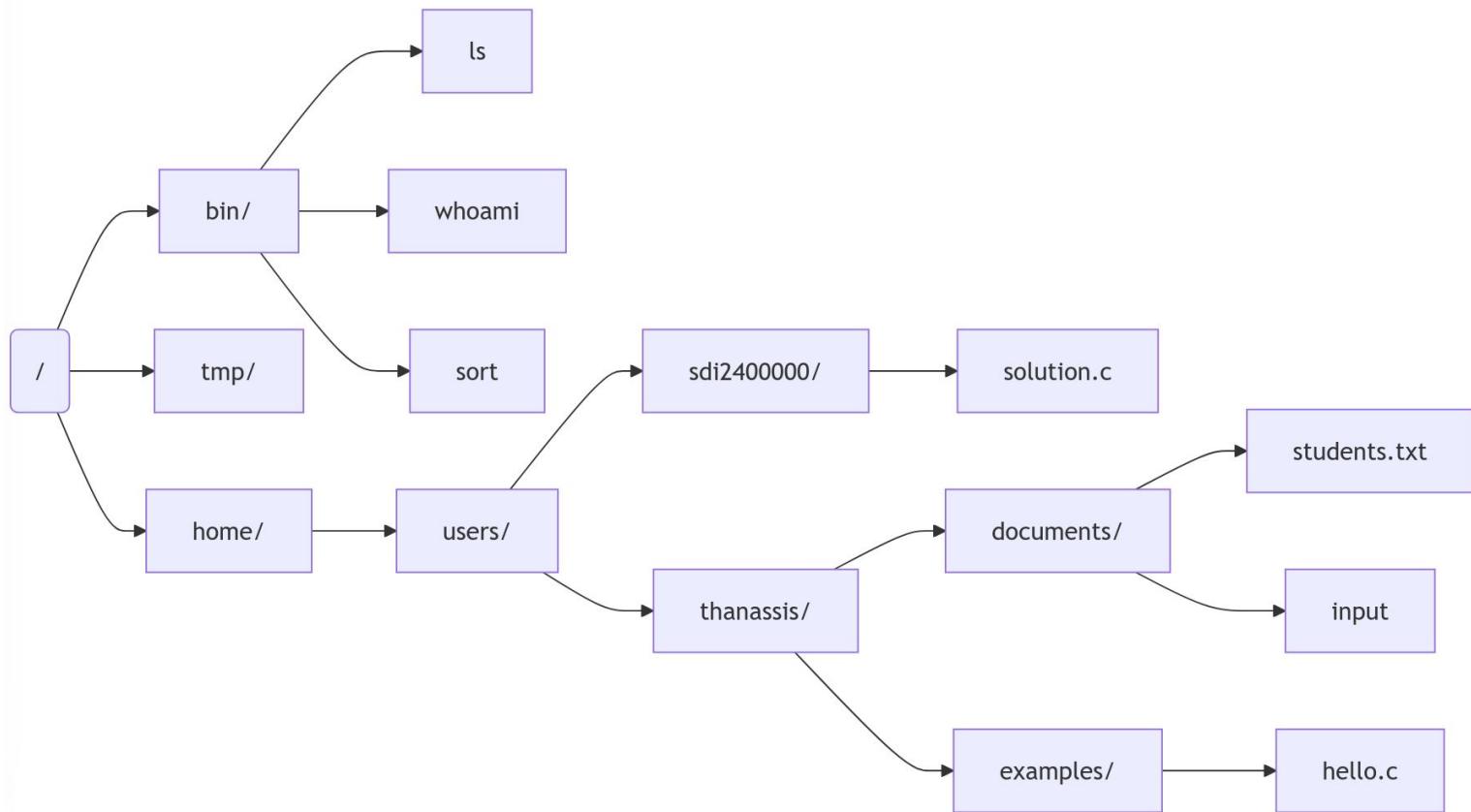
File (Αρχείο) είναι ένας πόρος για να καταγράφουμε δεδομένα σε έναν υπολογιστή. Συνήθως αποθηκεύεται στην δευτερεύουσα/μόνιμη μνήμη (πχ σκληρός δίσκος).

- Στο Linux σχεδόν τα πάντα είναι ένα αρχείο.
- Κάθε αρχείο:
 - Έχει ένα όνομα (**filename/basename**)
 - Βρίσκεται μέσα σε ένα συγκεκριμένο κατάλογο/φάκελο (**directory/folder**)
 - Έχει ένα μονοπάτι (**filepath**) που καθορίζει που βρίσκεται το αρχείο μέσα στο σύστημα αρχείων

Παράδειγμα: το filepath ενός αρχείου είναι

/home/users/thanassis/documents/students.txt, ο φάκελος μέσα στον οποίο βρίσκεται αυτό το αρχείο είναι ο /home/users/thanassis/documents ενώ το όνομα του αρχείου είναι students.txt. Το .txt στο τέλος του ονόματος λέγεται επέκταση (**extension**) και συνήθως περιγράφει τον τύπο του αρχείου.

Filesystem Hierarchy (Ιεραρχία Συστήματος Αρχείων)



Βασικές Εντολές Χειρισμού Αρχείων (1/2)

```
# Directory we are currently in  
thanassis@linux14:~/examples$ pwd  
/home/users/thanassis/examples  
# List files  
thanassis@linux14:~/examples$ ls  
hello.c  
# List files with details  
thanassis@linux14:~/examples$ ls -l  
total 4  
-rw-r--r-- 1 thanassis dep 62 Oct  5 16:04 hello.c  
# Print contents of file  
thanassis@linux14:~/examples$ cat hello.c  
#include <stdio.h>  
  
int main() {  
    printf("Hello world\n");  
}  
# Create `foo` directory  
thanassis@linux14:~/examples$ mkdir foo  
thanassis@linux14:~/examples$ ls  
foo  hello.c  
# New directory is empty by default  
thanassis@linux14:~/examples$ ls foo
```

Βασικές Εντολές Χειρισμού Αρχείων (2/2)

```
# Change directory to the newly created one
thanassis@linux14:~/examples$ cd foo
# List files within the directory
thanassis@linux14:~/examples/foo$ ls
# Copy file from parent `..` directory to the current one `..`
thanassis@linux14:~/examples/foo$ cp ../hello.c .
thanassis@linux14:~/examples/foo$ ls
hello.c
# Check the file contents are what we expect
thanassis@linux14:~/examples/foo$ cat hello.c
#include <stdio.h>

int main() {
    printf("Hello world\n");
}
# Remove the file
thanassis@linux14:~/examples/foo$ rm hello.c
# Go to the parent directory
thanassis@linux14:~/examples/foo$ cd ..
thanassis@linux14:~/examples$ rmdir foo
thanassis@linux14:~/examples$ ls
hello.c
```

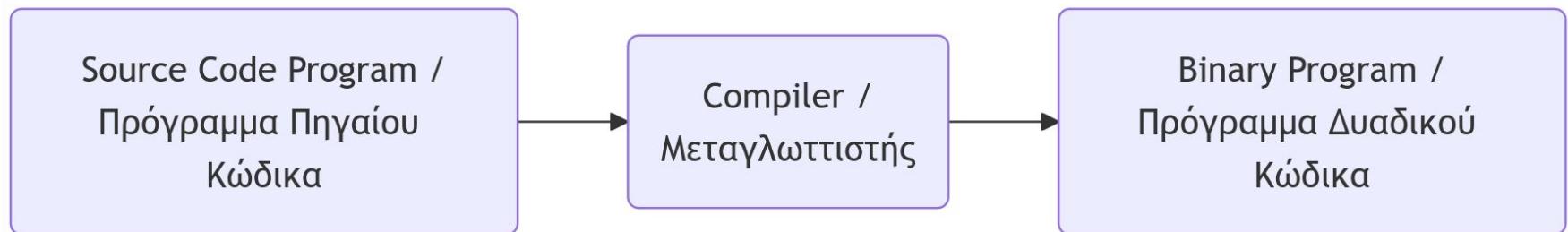
Coreutils (Βασικά Βοηθητικά Προγράμματα)

Τα περισσότερα Linux distros έρχονται με μια σουίτα προγραμμάτων που λέγονται [coreutils](#).

- Αυτά τα προγράμματα είναι ως επί το πλείστον γραμμένα σε C.
- Υπάρχουν και άλλα βασικά προγράμματα που δεν κοιτάξαμε (στα coreutils ή εκτός) τα οποία προτείνουμε να εξερευνήσετε
 - π.χ. `find, grep, sort, df -h, du -sh, wc -l, file, which, basename, dirname, ping, curl` κ.ο.κ.
- Στο live session μιλήσαμε λίγο και για `apt` (πως προσθέτουμε καινούρια προγράμματα) και command line editors (πως αλλάζουμε αρχεία από την κονσόλα). Περισσότερα στα εργαστήρια.

Compilers (Μεταγλωττιστές)

Compiler (μεταγλωττιστής) είναι ένα πρόγραμμα που μετατρέπει εντολές μιας γλώσσας προγραμματισμού σε κώδικα μηχανής ώστε να μπορεί να διαβαστεί και να τρέξει από τον υπολογιστή.



Η παραπάνω εικόνα είναι προσεγγιστική - λείπουν κάποιες λεπτομέρειες που θα προσθέσουμε σε επόμενες διαλέξεις. Στο μάθημα θα χρησιμοποιήσουμε τον [GNU C Compiler](#) ή αλλιώς gcc.

Ας κάνουμε compile το Hello World!

```
/* File: helloworld.c */

#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

Ας κάνουμε compile το Hello World!

```
thanassis@linux14:~/examples$ gcc hello.c
```

```
thanassis@linux14:~/examples$ ls
```

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

```
thanassis@linux14:~/examples$ ./a.out
```



```
Hello world
```

```
thanassis@linux14:~/examples$ file a.out
```

```
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter  
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=52fa5999c10d767a5ff30f662346478333de74bf, for GNU/Linux 3.2.0, not  
stripped
```

```
thanassis@linux14:~/examples$ gcc -o hello hello.c
```

```
thanassis@linux14:~/examples$ ./hello
```

```
Hello world
```

Ανάλυση του Hello World 1/4

```
/* File: helloworld.c */  
  
#include <stdio.h>  
  
int main() {  
    printf("Hello world\n");  
    return 0;  
}
```

Σχόλια (comments): κείμενο του προγραμματιστή που συνοδεύει τον κώδικα για να τον κάνει περισσότερο σαφή για τρίτους ή και εμάς τους ίδιους, ειδικά αν έχει περάσει καιρός από τότε που γράψαμε τον κώδικα :)

Περιέχεται ανάμεσα στα /* και */

Ή μπορεί να είναι σε μία γραμμή με //:

// single line comment

Ανάλυση του Hello World 2/4

```
/* File: helloworld.c */

#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

#include Directive (Οδηγία): Με την οδηγία `#include <stdio.h>` ο μεταγλωττιστής συμπεριλαμβάνει (include) τα περιεχόμενα του αρχείου `stdio.h` (standard input output) στον κώδικα του προγράμματος. Το αρχείο `stdio.h` περιέχει τις βασικές (standard) δηλώσεις των συναρτήσεων με τις οποίες γίνεται εμφάνιση δεδομένων στην οθόνη (output) και εισαγωγή δεδομένων από το πληκτρολόγιο (input).

Ανάλυση του Hello World 3/4

```
/* File: helloworld.c */
```

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

```
int main() {
```

```
    printf("Hello world\n");
```

```
    return 0;
```

```
}
```

printf: Η συνάρτηση βιβλιοθήκης printf δηλώνεται μέσα στο αρχείο stdio.h (για αυτό κάνουμε #include) και επιτρέπει την εκτύπωση του αλφαριθμητικού "Hello world\n". Ο χαρακτήρας '\n' δημιουργεί μια νέα γραμμή μετά την εμφάνιση του μηνύματος στην οθόνη.

Ανάλυση του Hello World 4/4 - Όνομα Συνάρτησης

- Ένα πρόγραμμα C ορίζεται από ένα σύνολο **συναρτήσεων** (next time).

```
int main() {
```

```
...
```

```
    return 0;
```

```
}
```

Προκειμένου να μπορούμε να το τρέξουμε, πρέπει να έχει **ακριβώς μία** συνάρτηση **main**, η οποία καλείται πρώτη όταν αρχίσουμε να τρέχουμε το πρόγραμμα.

Ανάλυση του Hello World 4/4 - Επιστροφή Συνάρτησης

- Ένα πρόγραμμα C ορίζεται από ένα σύνολο **συναρτήσεων** (next time).

```
int main() {  
    ...  
    return 0;  
}
```

Η εντολή `return 0` επιστρέφει την τιμή της συνάρτησης όταν αποτιμηθεί. Η τιμή που επιστρέφει η `main` είναι επίσης και το **exit code** του προγράμματος, δηλαδή η τιμή που δείχνει αν το πρόγραμμα ολοκληρώθηκε με επιτυχία ή όχι. Τρέχοντας `echo $?` σε ένα Linux shell μπορούμε να δούμε την τιμή με την οποία επέστρεψε το πρόγραμμα. Τιμές διάφορες του 0 σημαίνουν ότι το πρόγραμμα **απέτυχε**.

Ανάλυση του Hello World 4/4 - Εντολές Συνάρτησης

- Ένα πρόγραμμα C ορίζεται από ένα σύνολο **συναρτήσεων** (next time).

```
int main() {  
    printf(...);  
    return 0;  
}
```

Οι εντολές (statements) της συνάρτησης περιέχονται μέσα σε άγκιστρα '{ }' και η κάθε μία τελειώνει με ';' (semicolon / ελληνικό ερωτηματικό).

Διάλεξη 2 - Μνήμη και Μεταβλητές

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Την Προηγούμενη Φορά

- Αρχιτεκτονική Υπολογιστικών Συστημάτων
- Linux και Γραμμή Εντολών
- Βασικές Εντολές
- Μεταγλωττιστές
- Hello World

Bits & Bytes

Bit (binary digit): η μικρότερη μονάδα πληροφορίας σε υπολογιστή - 0 ή 1.

1

0

Byte: ομάδα από bit που αποθηκεύονται σε ένα κελί μνήμης

- Συνήθως κάθε byte είναι 8-bit (octet / οκτάδα)

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

Πόσα διαφορετικά 8-bit bytes μπορούμε να έχουμε;

Η Μνήμη Οργανώνεται σε Bytes

Το μέγεθος της μνήμης μετράται σε Bytes:

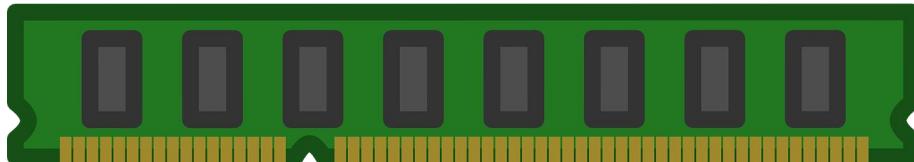
- **1 KB (KiloByte) = 1.000 Bytes**
- **1 MB (MegaByte) = 1.000.000 Bytes**
- **1 GB (GigaByte) = 1.000.000.000 Bytes**

Μνήμη με
N Bytes

Byte 0
Byte 1
Byte 2
...
Byte N-1
Byte N

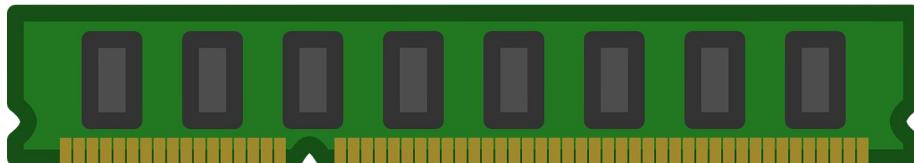
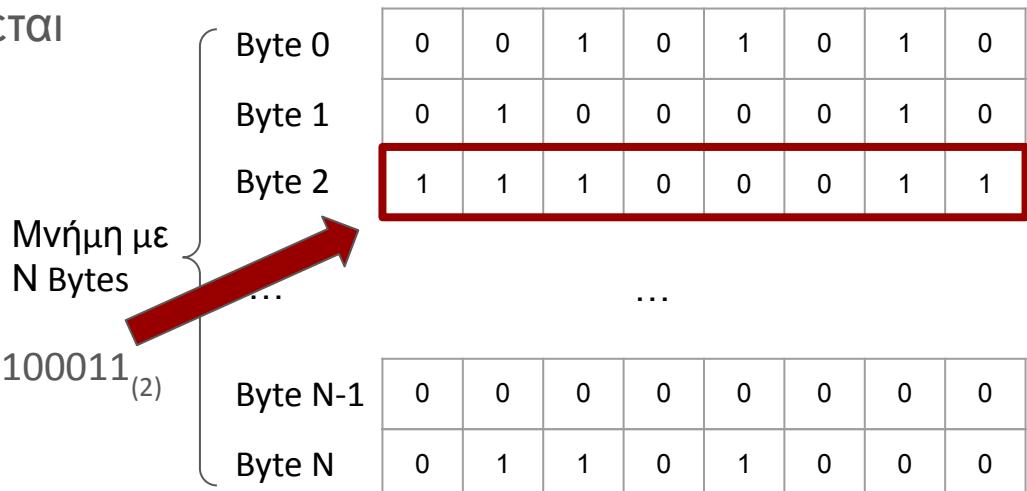
0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
1	1	1	0	0	0	1	1
...							

0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0



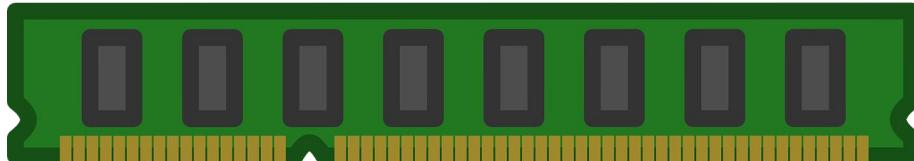
Διεύθυνση ενός Κελιού Μνήμης

Η θέση ενός κελιού στην μνήμη λέγεται
διεύθυνση (address).



Κύρια και Δευτερεύουσα Μνήμη

- Η **δευτερεύουσα μνήμη** (*secondary memory*) αποθηκεύει δεδομένα μόνιμα, δηλαδή σε υλικό που τα διατηρεί ακόμα και χωρίς παροχή ρεύματος (σκληροί δίσκοι, flash drives, κτλ)
- Η **κύρια μνήμη** (*primary memory*) αποθηκεύει δεδομένα προσωρινά αλλά επιτρέπει στον επεξεργαστή την άμεση πρόσβασή και επεξεργασία τους



Γενικά όποτε αναφερόμαστε σε "μνήμη" εννοούμε την κύρια μνήμη εκτός αν διευκρινίσουμε περαιτέρω!

Δυαδικοί Αριθμοί

Ένας **δυαδικός αριθμός** εκφράζεται με μια ακολουθία από δύο σύμβολα: το 0 και το 1. Ισοδύναμα, λέμε ότι ο αριθμός εκφράζεται με βάση το 2 / εκφράζεται στο δυαδικό σύστημα / έχει δυαδική αναπαράσταση.

Για παράδειγμα, έστω ο δυαδικός αριθμός: $101010_{(2)}$

Στο δεκαδικό σύστημα ο ίδιος αριθμός γράφεται:

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 42_{(10)}$$

Αντιστρέφοντας την διαδικασία μπορούμε να βρούμε την δυαδική αναπαράσταση οποιουδήποτε δεκαδικού αριθμού.

Δεκαεξαδικοί Αριθμοί και άλλες Βάσεις

Ένας δεκαεξαδικός αριθμός εκφράζεται με μια ακολουθία συνδυασμών από 16 σύμβολα:

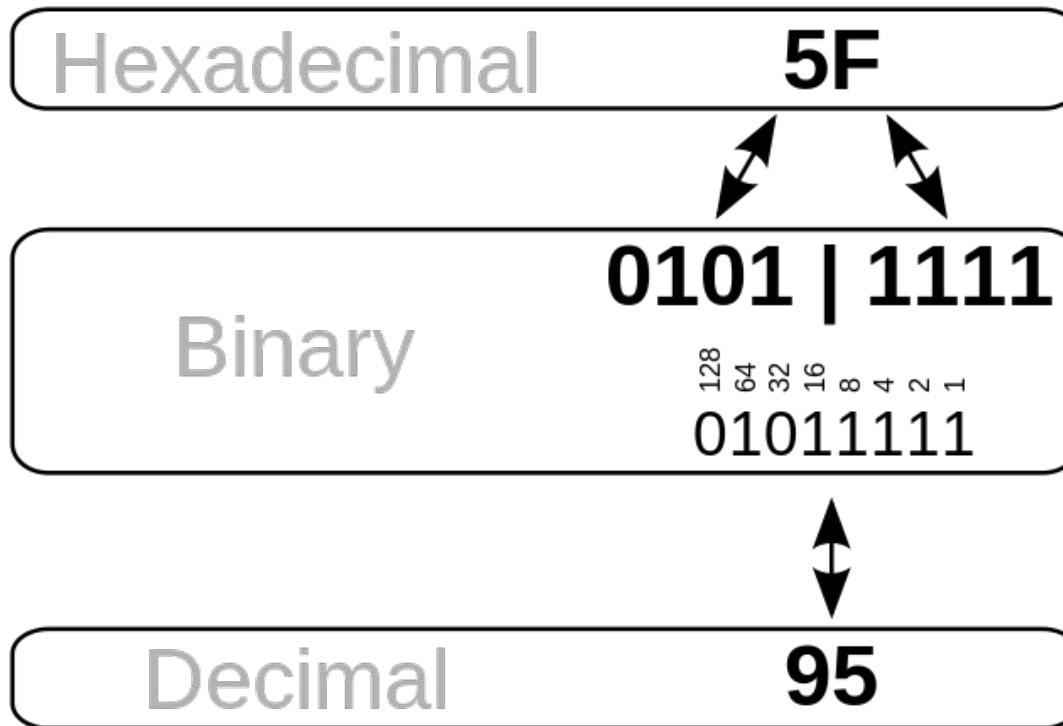
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Η μετατροπή ανάμεσα σε συστήματα γίνεται με τον ίδιο τρόπο:

$$2A_{(16)} = 2 \times 16^1 + A \times 16^0 = 42_{(10)}$$

Πόσα ψηφία χρειαζόμαστε στο δεκαεξαδικό σύστημα για να εκφράσουμε ένα byte; Είναι το ίδιο με το δεκαδικό;

Το ίδιο Byte - Διαφορετικά ψηφία σε κάθε βάση



Μεταβλητές και Χρήση Μνήμης

Δήλωση Μεταβλητής (Variable Declaration)

Μεταβλητή είναι ένα τμήμα της μνήμης με συγκεκριμένο όνομα.

Η μεταβλητή για να χρησιμοποιηθεί πρέπει να έχει δηλωθεί με κάποιον τύπο. Μορφή:

τύπος όνομα;

Ο τύπος (type) της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για τα περιεχόμενα

Το όνομα (name) της μεταβλητής κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης που θα την αποθηκεύσει

Δήλωση Μεταβλητής (Variable Declaration)

Μεταβλητή είναι ένα τμήμα της μνήμης με συγκεκριμένο όνομα.

Η μεταβλητή για να χρησιμοποιηθεί πρέπει να έχει δηλωθεί με κάποιον τύπο.

Ο τύπος (type) της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για τα περιεχόμενα

int x;

Το όνομα (name) της μεταβλητής κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης που θα την αποθηκεύσει

π.χ. 4 bytes για την x ξεκινώντας από το 0

Byte 0	0	0	1	0	1	0	1	0
Byte 1	0	1	0	0	0	0	1	0
Byte 2	1	1	1	0	0	0	1	1
Byte 3	1	1	1	0	0	0	1	1
...	...							
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

Δήλωση Μεταβλητής (Variable Declaration)

Μεταβλητή είναι ένα τμήμα της μνήμης με συγκεκριμένο όνομα.

Η μεταβλητή για να χρησιμοποιηθεί πρέπει να έχει δηλωθεί με κάποιον τύπο.

char c;

Ο τύπος (type) της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για τα περιεχόμενα

Το όνομα (name) της μεταβλητής κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης θα την αποθηκεύσει

Byte 0	0	0	1	0	1	0	1	0
Byte 1	0	1	0	0	0	0	1	0
Byte 2	1	1	1	0	0	0	1	1
Byte 3	1	1	1	0	0	0	1	1
...	...							
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

π.χ. 1 byte για την c ξεκινώντας από το N-1

Ερώτηση: ποια εντολή του Linux μας επιτρέπει να μάθουμε για
ένα πρόγραμμα στο σύστημά μας;

Ερμηνεία Δυαδικών Ψηφίων: Χαρακτήρες ASCII

Τα 8bits ενός char μπορεί να ερμηνευθούν ως χαρακτήρας κειμένου με την αντιστοίχιση του πίνακα [ASCII](#) (man ascii).

π.χ., ο αριθμός $67_{(10)} = 43_{(16)}$

αντιστοιχεί στο γράμμα 'C'

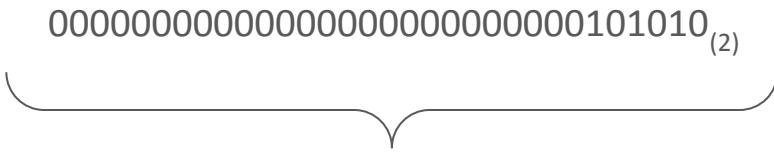
Πόσους διαφορετικούς χαρακτήρες έχουμε στο σύστημα ASCII; Γιατί;

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0' (null character)	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
013	11	0B	VT '\v' (vertical tab)	113	75	4B	K
014	12	0C	FF '\f' (form feed)	114	76	4C	L
015	13	0D	CR '\r' (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V

Manual page ascii(7) line 13/122 33% (press h for help or q to quit)

Αναπαράσταση Ακεραίων στην Μνήμη

Ο τύπος int που αναπαριστά ακεραίους απαιτεί **συνήθως** 4 byte από τον μεταγλωττιστή



000000000000000000000000101010₍₂₎

4 bytes = 32 bit

Για την μετατροπή τους σε δεκαδικό: $0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + \dots = 42_{(10)}$

Πόσους διαφορετικούς ακεραίους μπορούμε να αναπαραστήσουμε με 32 bit;

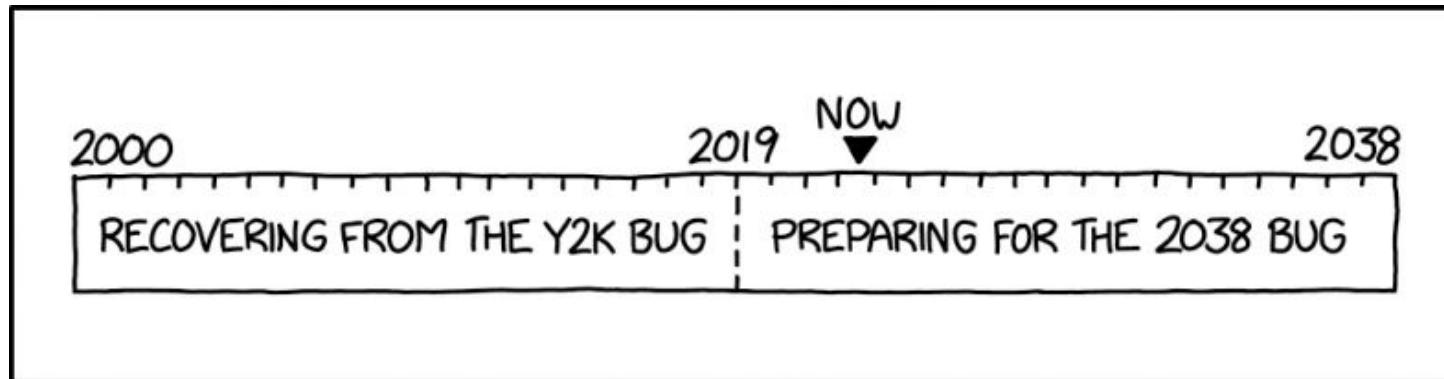
Υπάρχει κάποιο πιθανό πρόβλημα με αυτήν την αναπαράσταση;

Προβλήματα Αναπαράστασης Ακεραίων

1. $2^{32} \sim 4$ δις ακέραιοι ίσως να μην αρκούν για το πρόγραμμά μας
 - Ευτυχώς η C προσφέρει τύπους με μεγαλύτερο μέγεθος (π.χ., 64bit)
2. Υπάρχουν θετικοί και αρνητικοί ακέραιοι. Πως αναπαριστούμε το γεγονός ότιένας αριθμός είναι αρνητικός στην δυαδική αναπαράσταση; Τι κάνουμε με το πρόσημο;
 - Συμπλήρωμα του 2

Προβλήματα Αναπαράστασης Ακεραίων

1. $2^{32} \sim 4$ δις ακέραιοι ίσως να μην αρκούν για το πρόγραμμά μας
 - ο Ευτυχώς η C προσφέρει τύπους με μεγαλύτερο μέγεθος (π.χ., 64bit)



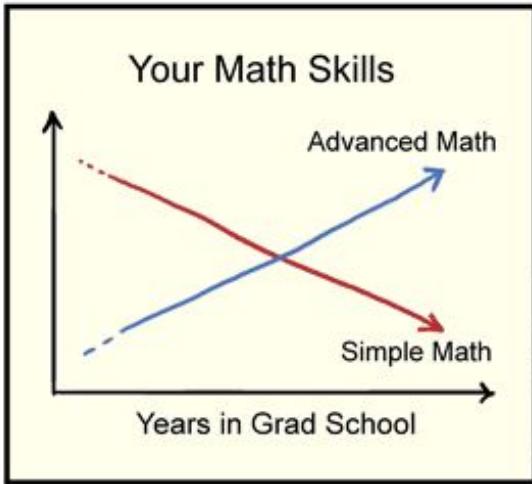
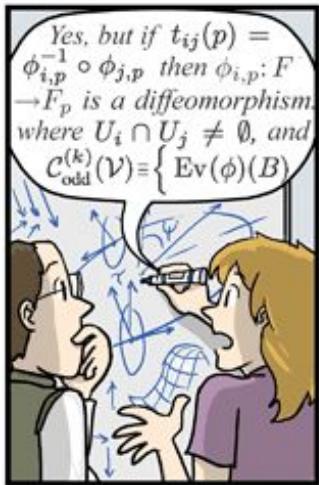
REMINDER: BY NOW YOU SHOULD HAVE FINISHED YOUR Y2K RECOVERY AND BE SEVERAL YEARS INTO 2038 PREPARATION.

Συμπλήρωμα του 2 (Two's complement)

Βασική ιδέα: να αναπαραστήσουμε το πρόσημο με το σημαντικότερο bit του αριθμού ($0 \rightarrow +$, $1 \rightarrow -$). Έστω ένας θετικός δυαδικός αριθμός N . Προκειμένου να πάρουμε την αναπαράσταση του $-N$, ακολουθούμε τα βήματα:

1. Αντίστρεψε (flip) όλα τα bit του N .
2. Πρόσθεσε 1.

Π.χ.: έστω $N = 11_{(10)} = 01011_{(2)}$, τότε το $-N$ είναι: $10100_{(2)} + 1_{(2)} = 10101_{(2)}$



Τύποι Μεταβλητών

Τύπος	Συνηθισμένο μέγεθος (bytes)	Εύρος τιμών (min-max)	Παράδειγμα Τιμής
char	1	-128 ... 127	'B', 0x42
short int	2	-32.768 ... 32.767	42
int	4	-2.147.483.648 ... 2.147.483.647	42
long int	4	-2.147.483.648 ... 2.147.483.647	42L
float	4	Μικρότερη θετική τιμή: 1.17×10^{-38} Μεγαλύτερη θετική τιμή: 3.4×10^{38}	42.42F
double	8	Μικρότερη θετική τιμή: 2.2×10^{-308} Μεγαλύτερη θετική τιμή: 1.8×10^{308}	42.42
long double	8, 10, 12, 16		42.42L
unsigned char	1	0 ... 255	'B', 0x42
unsigned short int	2	0 ... 65535	42
unsigned int	4	0 ... 4.294.967.295	42
unsigned long int	4	0 ... 4.294.967.295	42LU

Ανάθεση σε Μεταβλητή (Variable Assignment)

Ανάθεση σε μια μεταβλητή μπορεί να γίνει κατά τον ορισμό της:

```
int x = 42;
```

Ή μετά τον ορισμό της:

```
int x;
```

```
x = 42;
```

Ή με δεκαεξαδικό τρόπο:

```
int x = 0x2A;
```

Περιεχόμενο της x
πριν την ανάθεση

Byte 0	0	0	1	0	1	0	1	0
Byte 1	0	1	0	0	0	0	1	0
Byte 2	1	1	1	0	0	0	1	1
Byte 3	1	1	1	0	0	0	1	1
...
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

Ανάθεση σε Μεταβλητή (Variable Assignment)

Ανάθεση σε μια μεταβλητή μπορώ να γίνει κατά τον ορισμό της:

```
int x = 42;
```

Ή μετά τον ορισμό της:

```
int x;
```

```
x = 42;
```

Ή με δεκαεξαδικό τρόπο:

```
int x = 0x2A;
```

Γίνεται και σε οκταδικό: x = 052;

Περιεχόμενο της x
μετά την ανάθεση

Byte 0	0	0	0	0	0	0	0	0
Byte 1	0	0	0	0	0	0	0	0
Byte 2	0	0	0	0	0	0	0	0
Byte 3	0	0	1	0	1	0	1	0
...	...							
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

Υπερχείλιση Ακεραίων (Integer Overflow)

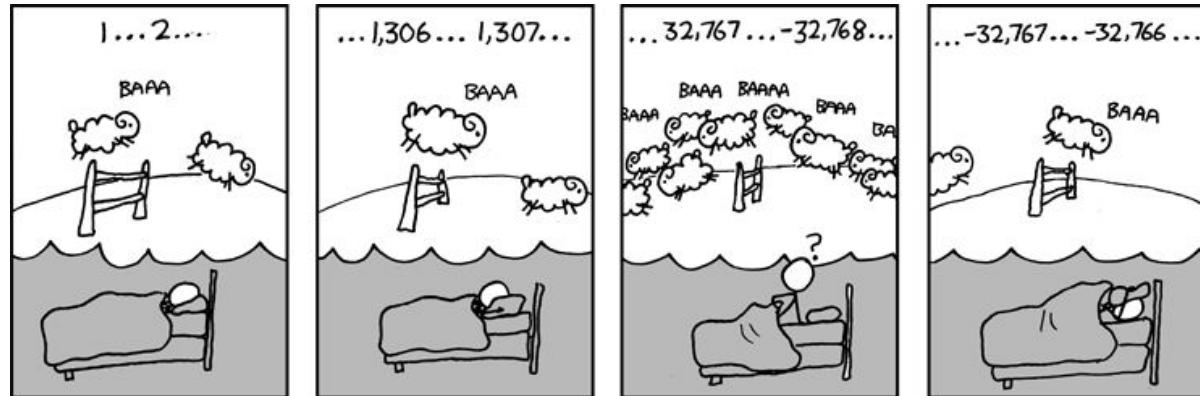
Τι θα τυπώσει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    printf("%d\n", 2000000000 + 2000000000);
    return 0;
}
```

```
$ ./overflow
-294967296
```

Τι συνέβη; Πως το διορθώνουμε;



Ακέραιοι με συγκεκριμένη ακρίβεια

Είπαμε ότι το μέγεθος ενός int είναι συνήθως 4 bytes - υπάρχει τρόπος να είμαστε βέβαιοι;

- Με την χρήση της sizeof μπορούμε να βρούμε το μέγεθος ενός τύπου - περισσότερα σε μελλοντικές διαλέξεις
- Με την χρήση της βιβλιοθήκης <stdint.h> μπορούμε να δηλώσουμε ακεραίους με την επιθυμητή ακρίβεια:

`int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t`

Η Συνάρτηση printf

Η συνάρτηση printf() χρησιμοποιείται για το τύπωμα δεδομένων στο αρχείο εξόδου stdout (standard output)

- Έχει μία μεταβλητή λίστα παραμέτρων:
 - a. Η πρώτη παράμετρος είναι ένα αλφαριθμητικό μορφοποίησης (format string), δηλαδή μία ακολουθία χαρακτήρων μέσα σε διπλά εισαγωγικά (" ") η οποία καθορίζει τον τρόπο με τον οποίο θα τυπωθούν τα δεδομένα.
 - b. Οι επόμενες παράμετροι είναι προαιρετικές και, αν υπάρχουν, η printf() μπορεί να χρησιμοποιήσει τις τιμές τους
- Το αλφαριθμητικό μορφοποίησης (format string) μπορεί να περιέχει:
 - a. Απλούς χαρακτήρες (οι οποίοι εμφανίζονται όπως είναι στην οθόνη)
 - b. Ακολουθίες διαφυγής (Escape sequences)
 - c. Προσδιοριστικά μορφοποίησης (Format specifiers)

Ακολουθίες Διαφυγής (Escape Sequences)

Η ακολουθία διαφυγής αποτελείται από μία ανάστροφη κεκλιμένη (\) (backslash) και έναν χαρακτήρα

Escape Sequence	Σημασία
\n	Αλλαγή γραμμής, σαν το πλήκτρο Enter
\r	Επαναφορά του δροιμέα (cursor) στην αρχή της τρέχουσας γραμμής
\t	Τύπωμα ενός κενού ίσο με το tab, σαν το πλήκτρο Tab
\ \	Τύπωμα ανάστροφης κεκλιμένης (backslash)
\ "	Τύπωμα διπλών εισαγωγικών (double quotes) ("")
\xNN	Τύπωμα του χαρακτήρα που αντιστοιχεί σε NN σε δεκαεξαδικό
\a	Δημιουργία ηχητικού σήματος

Προσδιοριστικά Μορφοποίησης (Format Specifiers)

Τα προσδιοριστικά μορφοποίησης αποτελούνται από τον χαρακτήρα % ακολουθούμενο από έναν ή περισσότερους χαρακτήρες που προσδιορίζουν πως να γίνει η μορφοποίηση (πλήρης λίστα [εδώ](#))

FormatSpecifier	Σημασία
%c	Τύπωμα ASCII χαρακτήρα
%d ή %i	Τύπωμα ακεραίου
%u	Τύπωμα unsigned (μη-προσημασμένου) ακεραίου
%f	Τύπωμα float
%llu	Τύπωμα long long unsigned int
%%	Τύπωμα του χαρακτήρα %

Δηλώσεις Πολλών Μεταβλητών

Μεταβλητές του ίδιου τύπου μπορούν να δηλωθούν στην ίδια γραμμή, διαχωρισμένες με κόμμα (,):

```
int a;  
int b;  
int c;
```

Μπορεί να γραφτεί ισοδύναμα ως:

```
int a, b, c;
```

Δεσμευμένες Λέξεις (Reserved Keywords) στην C

Οι δεσμευμένες λέξεις έχουν προκαθορισμένο νόημα για τον μεταγλωτιστή και **δεν** επιτρέπεται να χρησιμοποιηθούν για τον ορισμό μεταβλητών ή συναρτήσεων.

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

Διάλεξη 3 - Συναρτήσεις

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Η Συνάρτηση printf

Η συνάρτηση printf() χρησιμοποιείται για το τύπωμα δεδομένων στο αρχείο εξόδου stdout (standard output)

- Έχει μία μεταβλητή λίστα παραμέτρων:
 - a. Η πρώτη παράμετρος είναι ένα αλφαριθμητικό μορφοποίησης (format string), δηλαδή μία ακολουθία χαρακτήρων μέσα σε διπλά εισαγωγικά (" ") η οποία καθορίζει τον τρόπο με τον οποίο θα τυπωθούν τα δεδομένα.
 - b. Οι επόμενες παράμετροι είναι προαιρετικές και, αν υπάρχουν, η printf() μπορεί να χρησιμοποιήσει τις τιμές τους
- Το αλφαριθμητικό μορφοποίησης (format string) μπορεί να περιέχει:
 - a. Απλούς χαρακτήρες (οι οποίοι εμφανίζονται όπως είναι στην οθόνη)
 - b. Ακολουθίες διαφυγής (Escape sequences)
 - c. Προσδιοριστικά μορφοποίησης (Format specifiers)

Ακολουθίες Διαφυγής (Escape Sequences)

Η ακολουθία διαφυγής αποτελείται από μία ανάστροφη κεκλιμένη (\) (backslash) και έναν χαρακτήρα

Escape Sequence	Σημασία
\n	Αλλαγή γραμμής, σαν το πλήκτρο Enter
\r	Επαναφορά του δροιμέα (cursor) στην αρχή της τρέχουσας γραμμής
\t	Τύπωμα ενός κενού ίσο με το tab, σαν το πλήκτρο Tab
\ \	Τύπωμα ανάστροφης κεκλιμένης (backslash)
\ "	Τύπωμα διπλών εισαγωγικών (double quotes) ("")
\xNN	Τύπωμα του χαρακτήρα που αντιστοιχεί σε NN σε δεκαεξαδικό
\a	Δημιουργία ηχητικού σήματος

Προσδιοριστικά Μορφοποίησης (Format Specifiers)

Τα προσδιοριστικά μορφοποίησης αποτελούνται από τον χαρακτήρα % ακολουθούμενο από έναν ή περισσότερους χαρακτήρες που προσδιορίζουν πως να γίνει η μορφοποίηση (πλήρης λίστα [εδώ](#))

FormatSpecifier	Σημασία
%c	Τύπωμα ASCII χαρακτήρα
%d ή %i	Τύπωμα ακεραίου
%u	Τύπωμα unsigned (μη-προσημασμένου) ακεραίου
%f	Τύπωμα float
%llu	Τύπωμα long long unsigned int
%%	Τύπωμα του χαρακτήρα %

Note: Στον Προγραμματισμό συνήθως υπάρχει συμμετρία

Παρένθεση που ανοίγει πρέπει να κλείνει

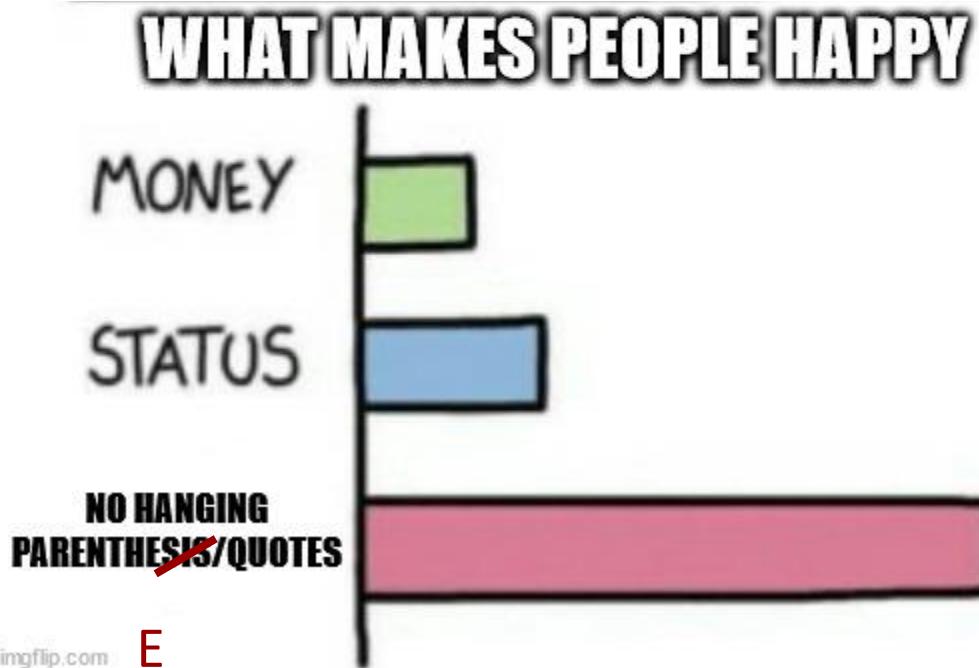
```
printf( " " )
```

Εισαγωγικά (quotes) που ανοίγουν πρέπει να κλείνουν

```
main( ) {
```

}

Αγκύλες (curly braces) που ανοίγουν πρέπει να κλείνουν



Διαβάζουμε τα error messages

```
$ gcc -o hello hello.c
hello.c: In function 'main':
hello.c:4:26: error: expected ';' before 'return'
  4 |   printf("Hello world\n")
      ^
      ;
  5 |   return 0;
      ~~~~~
```



Δηλώσεις Πολλών Μεταβλητών

Μεταβλητές του ίδιου τύπου μπορούν να δηλωθούν στην ίδια γραμμή, διαχωρισμένες με κόμμα (,):

```
int a;  
int b;  
int c;
```

Μπορεί να γραφτεί ισοδύναμα ως:

```
int a, b, c;
```

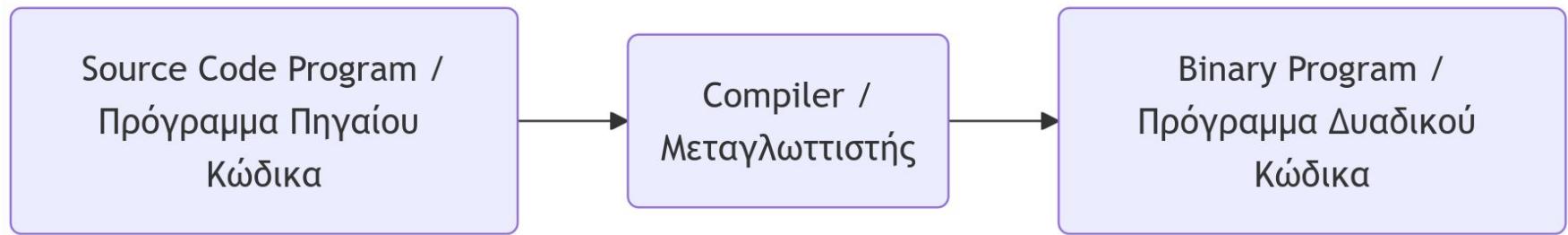
Δεσμευμένες Λέξεις (Reserved Keywords) στην C

Οι δεσμευμένες λέξεις έχουν προκαθορισμένο νόημα για τον μεταγλωτιστή και **δεν** επιτρέπεται να χρησιμοποιηθούν για τον ορισμό μεταβλητών ή συναρτήσεων.

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

Last Time: Compilers (Μεταγλωττιστές)

Compiler (μεταγλωττιστής) είναι ένα πρόγραμμα που μετατρέπει εντολές μιας γλώσσας προγραμματισμού σε κώδικα μηχανής ώστε να μπορεί να διαβαστεί και να τρέξει από τον υπολογιστή.



Η παραπάνω εικόνα είναι προσεγγιστική - λείπουν κάποιες λεπτομέρειες που θα προσθέσουμε σε επόμενες διαλέξεις. Στο μάθημα θα χρησιμοποιήσουμε τον [GNU C Compiler](#) ή αλλιώς gcc.

Ας κάνουμε compile το Hello World!

```
/* File: helloworld.c */

#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

Ας κάνουμε compile το Hello World!

```
thanassis@linux14:~/examples$ gcc hello.c
```

```
thanassis@linux14:~/examples$ ls
```

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

```
thanassis@linux14:~/examples$ ./a.out
```

```
Hello world
```

```
thanassis@linux14:~/examples$ file a.out
```

```
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter  
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=52fa5999c10d767a5ff30f662346478333de74bf, for GNU/Linux 3.2.0, not  
stripped
```

```
thanassis@linux14:~/examples$ gcc -o hello hello.c
```

```
thanassis@linux14:~/examples$ ./hello
```

```
Hello world
```

Ανάλυση του Hello World 1/4

```
/* File: helloworld.c */  
  
#include <stdio.h>  
  
int main() {  
    printf("Hello world\n");  
    return 0;  
}
```

Σχόλια (comments): κείμενο του προγραμματιστή που συνοδεύει τον κώδικα για να τον κάνει περισσότερο σαφή για τρίτους ή και εμάς τους ίδιους, ειδικά αν έχει περάσει καιρός από τότε που γράψαμε τον κώδικα :)

Περιέχεται ανάμεσα στα /* και */

Ή μπορεί να είναι σε μία γραμμή με //:

// single line comment

Ανάλυση του Hello World 2/4

```
/* File: helloworld.c */

#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

#include Directive (Οδηγία): Με την οδηγία `#include <stdio.h>` ο μεταγλωττιστής συμπεριλαμβάνει (include) τα περιεχόμενα του αρχείου `stdio.h` (standard input output) στον κώδικα του προγράμματος. Το αρχείο `stdio.h` περιέχει τις βασικές (standard) δηλώσεις των συναρτήσεων με τις οποίες γίνεται εμφάνιση δεδομένων στην οθόνη (output) και εισαγωγή δεδομένων από το πληκτρολόγιο (input).

Ανάλυση του Hello World 3/4

```
/* File: helloworld.c */
```

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

```
int main() {
```

```
    printf("Hello world\n");
```

```
    return 0;
```

```
}
```

printf: Η συνάρτηση βιβλιοθήκης printf δηλώνεται μέσα στο αρχείο stdio.h (για αυτό κάνουμε #include) και επιτρέπει την εκτύπωση του αλφαριθμητικού "Hello world\n". Ο χαρακτήρας '\n' δημιουργεί μια νέα γραμμή μετά την εμφάνιση του μηνύματος στην οθόνη.

Ανάλυση του Hello World 4/4 - Όνομα Συνάρτησης

- Ένα πρόγραμμα C ορίζεται από ένα σύνολο **συναρτήσεων**.

```
int main() {
```

```
...
```

```
    return 0;
```

```
}
```

Προκειμένου να μπορούμε να το τρέξουμε, πρέπει να έχει **ακριβώς** μία συνάρτηση **main**, η οποία καλείται πρώτη όταν αρχίσουμε να τρέχουμε το πρόγραμμα.

Ανάλυση του Hello World 4/4 - Επιστροφή Συνάρτησης

- Ένα πρόγραμμα C ορίζεται από ένα σύνολο **συναρτήσεων**.

```
int main() {  
    ...  
    return 0;  
}
```

Η εντολή `return 0` επιστρέφει την τιμή της συνάρτησης όταν αποτιμηθεί. Η τιμή που επιστρέφει η `main` είναι επίσης και το **exit code** του προγράμματος, δηλαδή η τιμή που δείχνει αν το πρόγραμμα ολοκληρώθηκε με επιτυχία ή όχι. Τρέχοντας `echo $?` σε ένα Linux shell μπορούμε να δούμε την τιμή με την οποία επέστρεψε το πρόγραμμα. Τιμές διάφορες του 0 σημαίνουν ότι το πρόγραμμα **απέτυχε**.

Ανάλυση του Hello World 4/4 - Εντολές Συνάρτησης

- Ένα πρόγραμμα C ορίζεται από ένα σύνολο **συναρτήσεων**.

```
int main() {  
    printf(...);  
    return 0;  
}
```

Οι εντολές (statements) της συνάρτησης περιέχονται μέσα σε άγκιστρα { } και η κάθε μία τελειώνει με ; (semicolon / ελληνικό ερωτηματικό).

Τι κρατάμε:

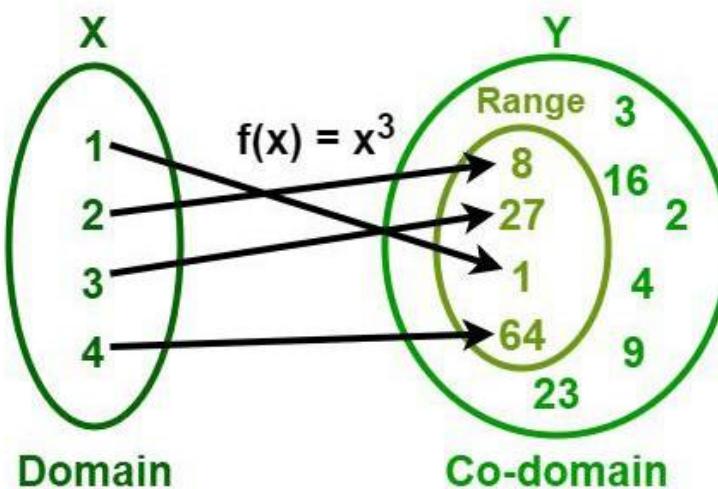
1. Ένα πρόγραμμα C είναι μια σειρά από συναρτήσεις
2. Η εκτέλεση του προγράμματος ξεκινά από την συνάρτηση main

Συνάρτησεις

Συνάρτηση: Μια αντιστοίχιση μεταξύ δύο συνόλων, που καλούνται σύνολο ορισμού και σύνολο τιμών, κατά την οποία κάθε ένα στοιχείο του πεδίου ορισμού αντιστοιχίζεται σε ένα και μόνο στοιχείο του πεδίου τιμών.

$$f(x) = x^3$$

$$f: \mathbb{Z} \rightarrow \mathbb{Z}$$



Πολυπαραμετρικές Συναρτήσεις

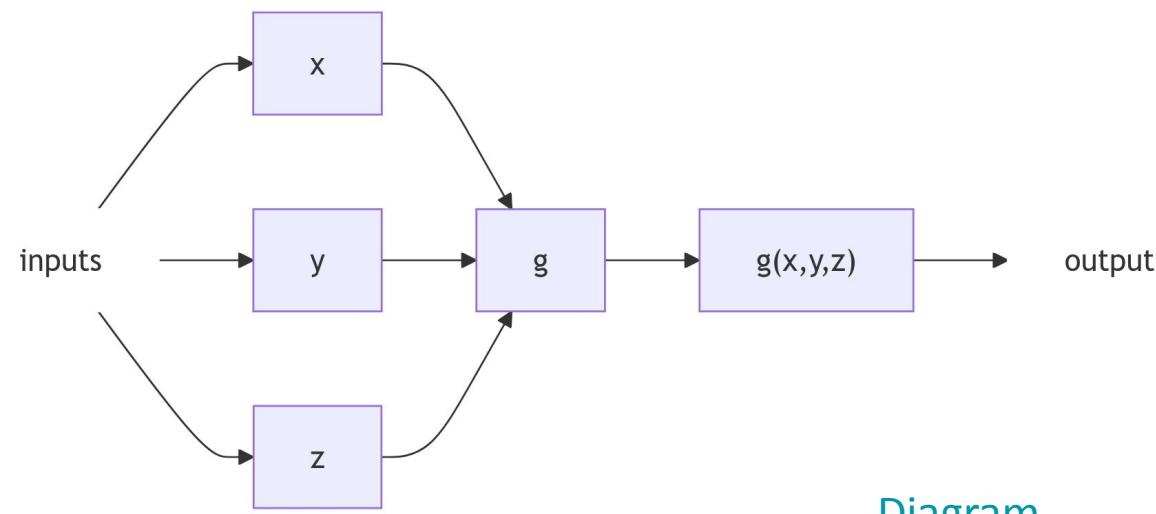
Συνάρτηση: Μια αντιστοίχιση μεταξύ δύο συνόλων, που καλούνται σύνολο ορισμού και σύνολο τιμών, κατά την οποία κάθε ένα στοιχείο του πεδίου ορισμού αντιστοιχίζεται σε ένα και μόνο στοιχείο του πεδίου τιμών.

$$g(x, y, z) = x^2 + y^2 + z^2 + 42$$

$$g: \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

$$h(x, y) = x + y + 1$$

$$h: \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}$$

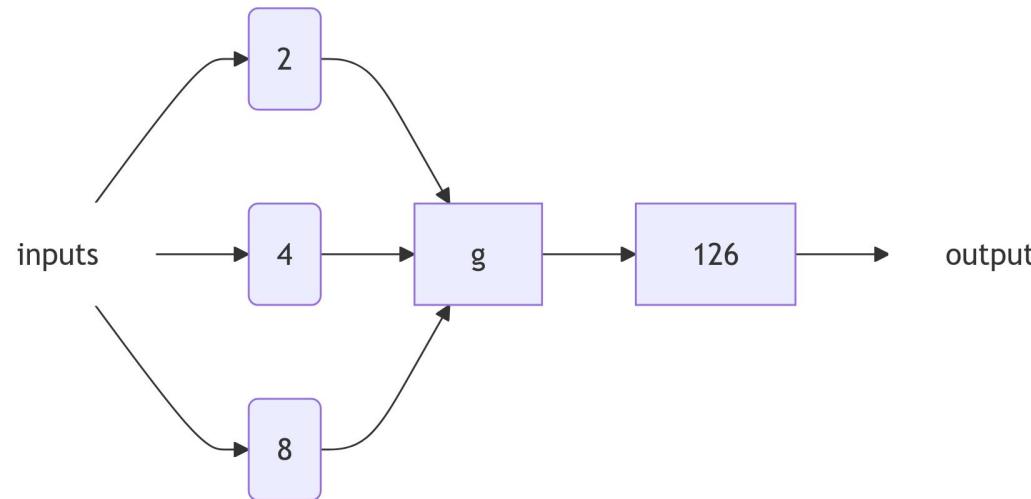


[Diagram](#)

Αποτίμηση Συναρτήσεων

$$g(x, y, z) = x^2 + y^2 + z^2 + 42$$

$$g(2, 4, 8) = 2^2 + 4^2 + 8^2 + 42 = 126$$



Συνάρτησεις στην C

Συνάρτηση είναι μια σειρά υπολογισμών που παίρνουν τις εισόδους της συνάρτησης και παράγουν την έξοδο.

Όπως και στα μαθηματικά, κάθε δεδομένο εισόδου και εξόδου της συνάρτησης έχει έναν **τύπο**, το σύνολο τιμών που μπορεί να πάρει. Για παράδειγμα, ο τύπος `int` στην C αντιπροσωπεύει τους ακεραίους (integers).

```
int g(int x, int y, int z) {  
    return x * x + y * y + z * z + 42;  
}
```

Ορισμός Συνάρτησης (Function Definition)

Το όνομα της συνάρτησης και τον τύπο της τιμής που επιστρέφει στην μορφή τύπος όνομα

- `int g` στο παράδειγμα. Αν καλέσουμε `g(...)` περιμένουμε ακέραιο αποτέλεσμα.

```
int g(int x, int y, int z) {  
    return x * x + y * y + z * z + 42;  
}
```

Ορισμός Συνάρτησης

Τα δεδομένα εισόδου ή ορίσματα της συνάρτησης εντός παρενθέσεων, επίσης της μορφής τύπος όνομα.

- int x, int y, int z είναι 3 ακέραια ορίσματα με ονόματα x, y και z.

```
int g(int x, int y, int z) {  
    return x * x + y * y + z * z + 42;  
}
```

Ορισμός Συνάρτησης

```
int g(int x, int y, int z) {  
    return x * x + y * y + z * z + 42;  
}
```

Το σώμα της συνάρτησης εντός των αγκυλών {} περιέχει τον υπολογισμό της τιμής της συνάρτησης.

- `return x * x + y * y + z * z + 42;` θα **επιστρέψει** ένα ακέραιο αποτέλεσμα.
- Οι εντολές στο σώμα της συνάρτησης διαχωρίζονται με semicolon (το ερωτηματικό ;).

Ορισμός Συνάρτησης

Το όνομα της συνάρτησης και τον **τύπο της τιμής που επιστρέφει** στην μορφή τύπος όνομα

- `int g(int x, int y, int z)` {

```
int g(int x, int y, int z) {  
    return x * x + y * y + z * z + 42;  
}
```

Τα **δεδομένα εισόδου** ή **ορίσματα** της συνάρτησης εντός παρενθέσεων, επίσης της μορφής τύπος όνομα.

- `int x, int y, int z` είναι 3 ακέραια ορίσματα με ονόματα `x`, `y` και `z`.

Το **σώμα** της συνάρτησης εντός των αγκυλών `{ }` περιέχει τον υπολογισμό της τιμής της συνάρτησης.

- `return x * x + y * y + z * z + 42;` θα **επιστρέψει** ένα ακέραιο αποτέλεσμα.
- Οι εντολές στο σώμα της συνάρτησης διαχωρίζονται με semicolon (το ερωτηματικό `;`).

Κλήση Συνάρτησης (Function Call)

```
int g(int x, int y, int z) {  
    return x * x + y * y + z * z + 42;  
}
```

```
int main(42, int y, int z) {  
    ...  
    int x = g(1, 2, 3);  
    ...  
}
```

Ποια η τιμή του x μετά την εκτέλεση αυτής της ανάθεσης;

Η κλήση της συνάρτησης γίνεται με το όνομά της και στην συνέχεια περνάμε τα ορίσματά της ανάμεσα σε παρενθέσεις χωρισμένα με ,

Pair Programming

Pair programming is a [software development](#) technique in which two [programmers](#) work together at one workstation. One, the *driver*, writes [code](#) while the other, the *observer* or *navigator*, [reviews](#) each line of code as it is typed in. The two programmers switch roles frequently.



Διάλεξη 4 - Live Coding, Git και Τελεστές

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Υπολογισμός Βαθμολογίας Πρωτοετών

- $50\% * \text{Τελική Εξέταση} + 30\% * \text{Ασκήσεις} + 20\% * \text{Εργαστήριο}$
- Έστω ότι χρησιμοποιούμε 3 ακεραίους για να αναπαραστήσουμε τα αποτελέσματα [0 - 100].
 - int final_exam
 - int homework
 - int lab
- Στα μαθηματικά:
 - $\text{grade}(\text{final_exam}, \text{homework}, \text{lab}) = \text{final_exam} \times 50\% + \text{homework} \times 30\% + \text{lab} \times 20\%$
- Έλεγχος ορθότητας: $\text{grade}(70, 80, 100) == 79?$
- Σε C;

Πρόγραμμα Υπολογισμού Βαθμολογίας 1/2

```
#include <stdio.h>

#include <stdlib.h>

// Compute grades using the class formula

int grade(int final_exam, int homework, int lab) {

    return final_exam * 50 / 100 + homework * 30 / 100 + lab * 20 / 100;
}

int main(int argc, char **argv) {

    if (argc != 4) {

        printf("Program needs to be called as `./prog final_exam homework lab`\n");

        return 1;
    }

    int final_exam = atoi(argv[1]);

    int homework = atoi(argv[2]);

    int lab = atoi(argv[3]);

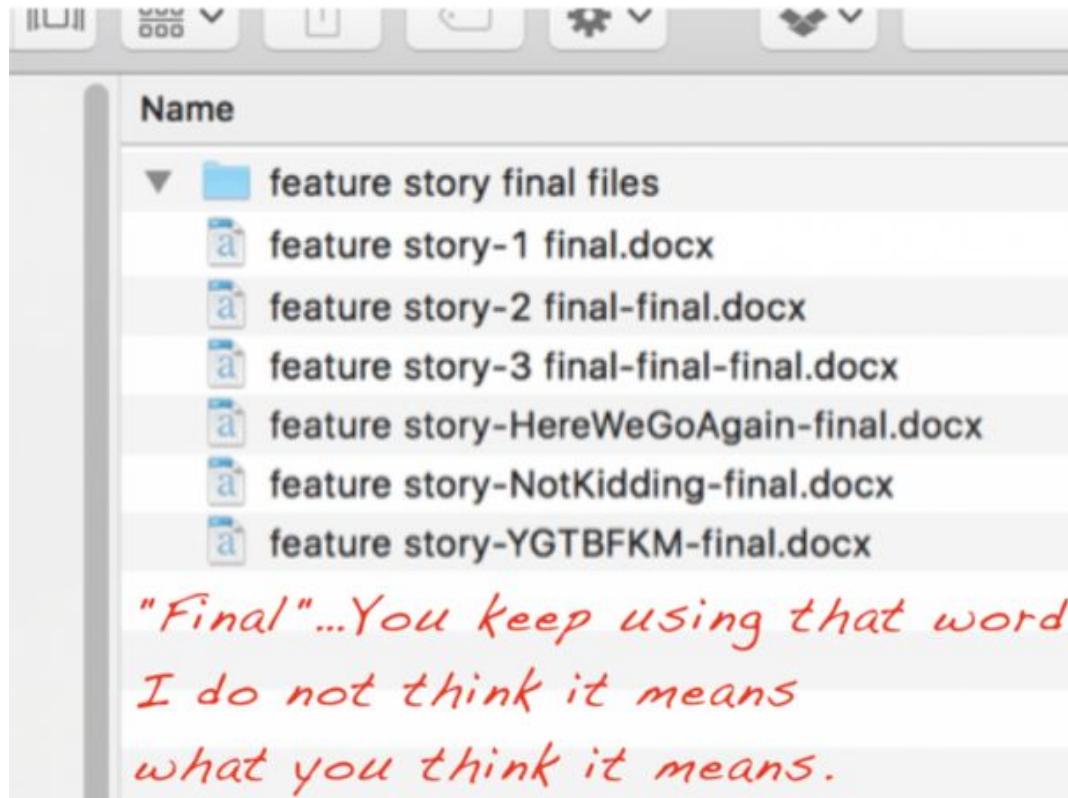
    printf("Grade: %d\n", grade(final_exam, homework, lab));

    return 0;
}
```

Πρόγραμμα Υπολογισμού Βαθμολογίας 1/2

- Η συνάρτηση `main` έχει μεταβλητό αριθμό ορισμάτων. Η παράμετρος `argc` λέει πόσα ορίσματα περάσαμε στο πρόγραμμα, ενώ η παράμετρος `argv` περιέχει το κείμενο που αντιστοιχεί σε κάθε ορίσμα.
 - `argv[1]` περιέχει το 1ο ορίσμα του προγράμματος, `argv[2]` το 2ο, κ.ο.κ.
- Η συνάρτηση βιβλιοθήκης (`stdlib.h`) αποτελεί μετατρέπει ένα ορίσμα από αλφαριθμητικό (ASCII) σε έναν ακέραιο (`integer`).
- Εάν σας ενοχλεί το πόσο "μαγική" φαίνεται η `main` και τα ορίσματά της, μην εξάπτεστε! Είναι μια από τις δυσκολότερες συναρτήσεις που θα συναντήσουμε στην C και τις επόμενες εβδομάδες θα χτίσουμε το υπόβαθρο για να γίνει πιο κατανοητή.

Version Control & Git



Version Control

- Ένα σύστημα που επιτρέπει να παρακολουθούμε όλες τις αλλαγές που γίνονται στον κώδικα και να κρατάμε ιστορικό αρχείο.

Γιατί να θέλουμε κάτι τέτοιο;;

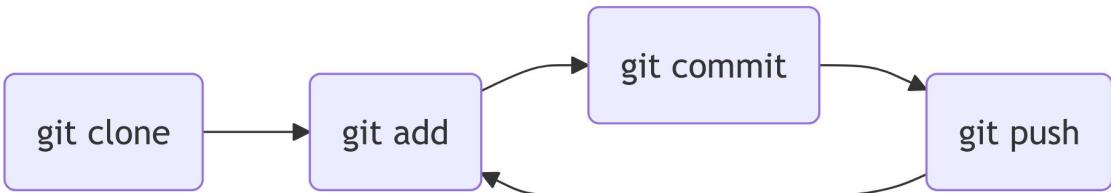
- Τα περισσότερα προγράμματα αλλάζουν με τον χρόνο - δεν υπάρχει "τελική μορφή"
- Κάνουμε αλλαγές και κρατάμε μόνο τα αρχεία που χρειαζόμαστε χωρίς να χάνουμε πληροφορία
- Μπορούμε να ανατρέξουμε σε οποιαδήποτε αλλαγή έγινε ανά πάσα στιγμή

Git

- Είναι το δημοφιλέστερο πρόγραμμα Version Control σήμερα (~90% των χρηστών/project)
- Πρώτη έκδοση γράφτηκε από τον Linus Torvalds το 2005
- Μια δημοφιλής και δωρεάν (για την ώρα) πλατφόρμα στην οποία μπορούμε να αποθηκεύσουμε τον κώδικά μας είναι το <https://github.com/>
- Θα το χρησιμοποιήσουμε για την γραφή και υποβολή των εργασιών μας

Git Development Cycle

Σε επίπεδο βασικής χρήσης (αυτό που χρειαζόμαστε για το μάθημα), η γραφή Κώδικα και η αποθήκευση με git περιλαμβάνει 4 βήματα:



THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.



Clone Repository

git clone: Αντιγράφει το repository (αποθετήριο) με όλα τα αρχεία σε έναν τοπικό φάκελο.

```
git clone git@github.com:progintro/hw0-barbouni-2005.git
```

- Αν ελέγξουμε τον φάκελο παρατηρούμε ότι έχει ένα README.md αρχείο.
- Τρέχοντας git status μέσα στον φάκελο μπορούμε να δούμε αν υπάρχουν καθόλου αλλαγές.
- Η παραπάνω εντολή θα δημιουργήσει έναν τοπικό φάκελο (αν δεν υπάρχει) με το όνομα hw0-barbouni-2005
- Τα βήματα που ακολουθούμε εδώ, προϋποθέτουν ότι έχετε φτιάξει ένα κλειδί για τον Github λογαριασμό σας και έχετε τελειώσει configuration όπως περιγράφεται στο Φυλλάδιο 1 του εργαστηρίου.

Προσθήκη Αρχείου

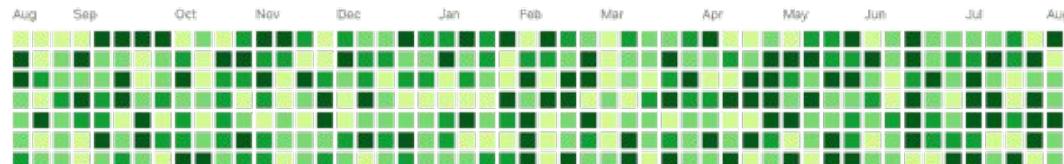
git add: Προσθέτει τα αρχεία στο staging area (προσωρινή λίστα) για μελλοντική αποθήκευση στο repository.

- Δημιούργησε ένα καινούριο αρχείο, π.χ., touch command/solution.txt.
- Τρέξε git status - θα παρατηρήσεις ότι το καινούριο αρχείο καταγράφεται ως untracked.
- Για να το προσθέσουμε: git add command/solution.txt
- Τρέξε git status ξανά - θα παρατηρήσεις ότι το αρχείο είναι έτοιμο να γίνει commit (καταχωρηθεί).

Καταχώρηση Αλλαγών

git commit: Αποθηκεύει τις αλλαγές στο τοπικό repository.

- Τρέξε git commit και δώσε ένα μήνυμα που περιγράφει την αλλαγή σου.
- Τρέξε git status και πάλι - τώρα το μόνο που μένει είναι να σώσουμε τις αλλαγές μας και εκτός του υπολογιστή μας.



**Looking for someone
who can commit**

Ανέβασμα Αλλαγών

`git push`: Ανεβάζει τις τοπικές αλλαγές στον απομακρυσμένο server (εξυπηρετητή).

- Έλεγχε ότι οι αλλαγές σου εμφανίζονται στο Github!

Κατέβασμα Αλλαγών

`git pull`: Κατεβάζει αλλαγές από τον απομακρυσμένο server (αν υπάρχουν).

- Χρήσιμο όταν έχεις πολλούς προγραμματιστές ή πολλούς υπολογιστές και δεν θέλεις να κάνεις `clone` κάθε φορά.
- Κάνε κάποιες αλλαγές στο repository από το GitHub UI και μετά τρέξε `git pull`.

Don't forget to add
your files before
committing &
pushing!



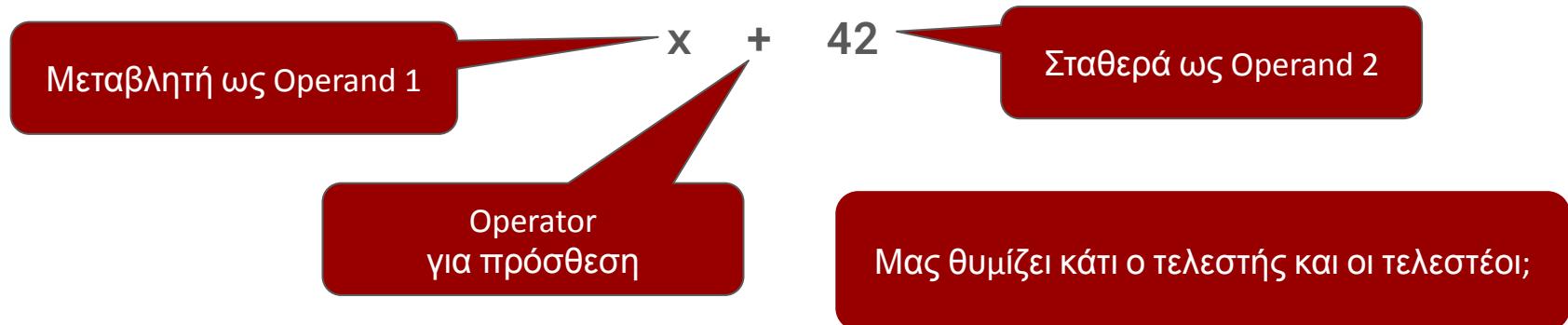
Τελεστές (Operators)

- Μοναδιαίοι (unary), Δυαδικοί (binary), Τριαδικοί (ternary)
- Αριθμητικοί, Συγκριτικοί, Λογικοί, Συνθήκης, Bitwise, Μετατροπής, Ανάθεσης
- Προτεραιότητα & Προσεταιριστικότητα

Τελεστής (Operator) και Τελεστέος (Operand)

Οι **τελεστές** στην C είναι σύμβολα που πραγματοποιούν μαθηματικούς, συγκριτικούς, δυαδικούς, υποθετικούς ή λογικούς υπολογισμούς.

Οι **τελεστέοι** στην C είναι τα αντικείμενα (μεταβλητές, σταθερές) πάνω στα οποία πραγματοποιούν τους υπολογισμούς οι τελεστές.



Πρόγραμμα - Συνάρτηση - Τελεστές

Το πρόγραμμα παίρνει κάποια δεδομένα εισόδου και υπολογίζει μια έξοδο

Το πρόγραμμα αποτελείται από συναρτήσεις που παίρνουν ορίσματα και υπολογίζουν μια έξοδο

Οι συναρτήσεις χρησιμοποιούν τελεστές και τελεστέους προκειμένου να υπολογίσουν μια έξοδο



It's turtles all the way down!

Κατηγοριοποίηση Τελεστών #1

Κατηγοριοποίηση με βάση το arity - πόσους τελεστέους έχουν:

- Μοναδιαίοι Τελεστές (Unary Operators): Ένα τελεστή (operand)
- Δυαδικοί Τελεστές (Binary Operators): Δύο τελεστές
- Τριαδικοί Τελεστές (Ternary Operators): Τρεις τελεστές

Στην συνέχεια θα δούμε κατηγοριοποίηση με βάση την χρήση του τελεστή και στην σειρά υπολογισμού

Παραδείγματα Τελεστών

Αριθμητικοί Τελεστές

Αριθμητικός Τελεστής	Ερμηνεία	Παραδείγματα
+	πρόσθεση	$40 + 2$ επιστρέφει 42 $40.0 + 2.0$ επιστρέφει 42.0
-	αφαίρεση	$44 - 2$ επιστρέφει 42 $44.0 - 2.0$ επιστρέφει 42.0
*	πολλαπλασιασμός	$42 * 2$ επιστρέφει 84 $42.0 * 2.0$ επιστρέφει 84.0
/	διαίρεση	$85.0 / 2.0$ επιστρέφει 42.5 $85 / 2$ επιστρέφει 42
%	υπόλοιπο	$7 \% 2$ επιστρέφει 1

- Σε τι κατηγορία θα εντάσσατε τους αριθμητικούς τελεστές;
- Τι τύπο θα είχαν οι τελεστές αν ήταν συναρτήσεις;

Συγκριτικούς Τελεστές (Comparison Operators)

Οι συγκριτικοί τελεστές επιτρέπουν την σύγκριση τελεστών. Το αποτέλεσμα είναι 0 (ψευδές) ή 1 (αληθές). Τι τύπου τελεστές είναι;

Σχεσιακός Τελεστής	Ερμηνεία	Παραδείγματα
<code>==</code>	Ίσοι τελεστέοι;	$42 == 0$ επιστρέφει 0 $42 == 42$ επιστρέφει 1
<code>!=</code>	Διαφορετικοί τελεστέοι;	$42 != 0$ επιστρέφει 1 $42 != 42$ επιστρέφει 0
<code>></code>	Αριστερός τελεστέος μεγαλύτερος του δεξιού;	$42 > 0$ επιστρέφει 1 $42 > 42$ επιστρέφει 0
<code><</code>	Αριστερός τελεστέος μικρότερος του δεξιού;	$42 < 0$ επιστρέφει 0 $42 < 44$ επιστρέφει 1
<code>>=</code>	Αριστερός τελεστέος μεγαλύτερος ή ίσος του δεξιού;	$42 >= 42$ επιστρέφει 1 $42 >= 43$ επιστρέφει 0
<code><=</code>	Αριστερός τελεστέος μικρότερος ή ίσος του δεξιού;	$42 <= 42$ επιστρέφει 1 $42 <= 41$ επιστρέφει 0

Λογικοί Τελεστές (Logical Operators)

Οι λογικοί τελεστές επιτρέπουν την αποτίμηση συνδυασμού αληθών και ψευδών τιμών.
Τι τύπου τελεστές είναι;

Λογικός Τελεστής	Ερμηνεία	Παραδείγματα
&&	Λογική σύζευξη (AND, \wedge) είναι και οι δύο τελεστέοι αληθείς;	$42 > 0 \ \&\& \ 2 > 3$ επιστρέφει 0 $42 > 0 \ \&\& \ 3 > 2$ επιστρέφει 1
	Λογική διάζευξη (OR, \vee) είναι ένας εκ των δύο τελεστέων αληθής;	$2 > 3 \ \ 42 > 0$ επιστρέφει 1 $2 > 3 \ \ 42 < 0$ επιστρέφει 0
!	Λογική άρνηση (NOT, \neg) είναι ο τελεστέος ψευδής;	$!(42 < 0)$ επιστρέφει 1 $!(42 > 0)$ επιστρέφει 0

Σημαντικό: για τους λογικούς τελεστές στην C, το μηδέν (0) σημαίνει “ψευδές”. Οποιαδήποτε άλλη μη μηδενική τιμή σημαίνει “αληθές”. Το 1, το 42 και το -5 είναι όλα εξίσου “αληθή”.

Bitwise Τελεστές (Bitwise Operators)

Οι **bitwise τελεστές** κάνουν μετατροπές στα *bit* των **ακέραιων** αριθμών. Τι τύπου είναι οι τελεστές;

Bitwise Τελεστής	Ερμηνεία	Παραδείγματα
&	Σύζευξη bit (bitwise AND)	0xFF & 0xF0 επιστρέφει 0xF0 0xFF & 0x00 επιστρέφει 0x00
	Διάζευξη bit (bitwise OR)	0xFF 0xF0 επιστρέφει 0xFF 0xF0 0x0F επιστρέφει 0xFF
^	Αποκλειστική διάζευξη bit (bitwise XOR)	0xFF ^ 0xFF επιστρέφει 0x00 0x00 ^ 0xFF επιστρέφει 0xFF
~	Άρνηση bit (bitwise NOT)	~0xF0 επιστρέφει 0x0F ~0x05 επιστρέφει 0xFA
<<	Ολίσθηση bit αριστερά (shift left). Shift N bit ισοδύναμο με πολλαπλασιασμό με 2^N	2 << 4 επιστρέφει 32 4 << 1 επιστρέφει 8
>>	Ολίσθηση bit δεξιά (shift right). Shift N bit ισοδύναμο με διαίρεση με 2^N	8 >> 1 επιστρέφει 4 32 >> 4 επιστρέφει 2

Διάλεξη 5 - Τελεστές και Εντολές

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Πως απομονώνουμε το πιο σημαντικό bit σε ένα byte;

A. byte & 0xFF

B. byte & 0x80

C .byte | 0xFF

D. byte | 0x80

Ποια η τιμή της παράστασης 0xbeef | 0xcafe0000;

A. 0xffffffff

B. 0xcafebef

C. 0xbeefcafe

D. 0xfaceb00c

Τελεστής Συνθήκης (Conditional Operator)

Ο τελεστής συνθήκης ελέγχει την συνθήκη και αν είναι αληθής επιστρέφει την πρώτη τιμή, αλλιώς επιστρέφει την δεύτερη. Γενική μορφή:

συνθήκη ? τιμή1 : τιμή2

Για παράδειγμα:

$(42 > 0) ? 8 : 16 \Rightarrow \text{επιστρέφει } 8$

Αντίστοιχα:

$(42 == 0) ? 8 : 16 \Rightarrow \text{επιστρέφει } 16$

Τι τύπου είναι ο τελεστής συνθήκης; Πως θα φτιάχναμε μια συνάρτηση max;

Τελεστής Μετατροπής (Cast Operator)

Ο τελεστής μετατροπής αλλάζει τον τύπο ενός τελεστέου. Γενική μορφή:

(τύπος)τελεστέος

Παραδείγματα:

(int)42.67 επιστρέφει 42

(char)67.8 επιστρέφει 'C'

(double)3 επιστρέφει 3.0

(double)3/4 επιστρέφει ??

Ψηφία μετά την
υποδιαστολή
αποκόπτονται, δεν
στρογγυλοποιούνται

Η παραπάνω μετατροπή λέγεται και explicit type conversion

Σιωπηρή Μετατροπή Τύπων (Implicit Type Conversion)

Σε περιπτώσεις μίξης τύπων σε έναν τελεστή, ο μεταγλωττιστής προσθέτει αυτόματα μετατροπές στον μεγαλύτερο των τελεστέων.

Παράδειγμα:

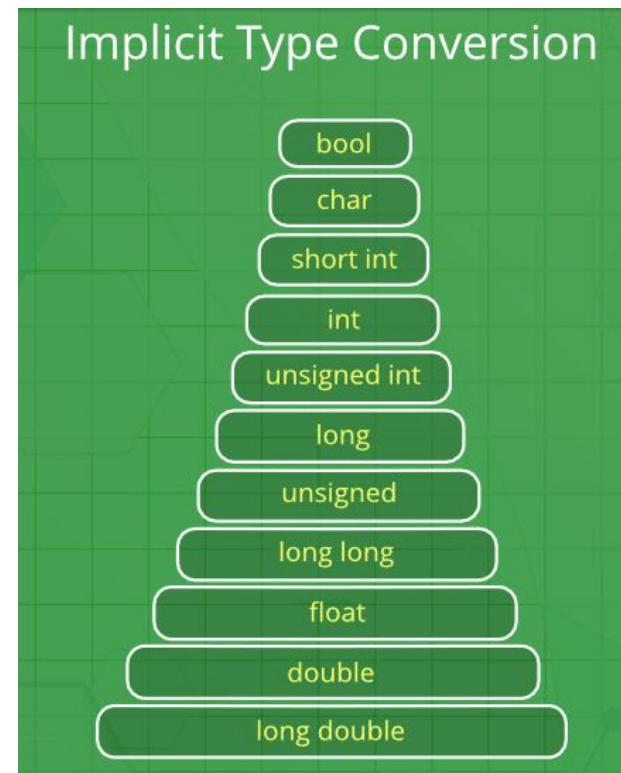
40 + 2.0

Είναι ισοδύναμο με:

(double)40 + 2.0

Ισοδύναμο με:

40.0 + 2.0



Τελεστής Ανάθεσης (Assignment Operator)

Ο τελεστής ανάθεσης παίρνει την τιμή του δεξιού τελεστέου (**rvalue**), την αναθέτει στην μεταβλητή του αριστερού τελεστέου (**lvalue**) και την επιστρέφει

Παράδειγμα:

`x = 42 // αναθέτει 42 στην x και επιστρέφει 42`

Μπορούμε να αναθέσουμε πολλές μεταβλητές ταυτόχρονα:

`x = y = 42`

Ή ισοδύναμα:

`x = (y = 42)`

Συνδυαστικοί Τελεστές Ανάθεσης

Οι **συνδυαστικοί τελεστές ανάθεσης** επιτρέπουν να κάνουμε αριθμητικές πράξεις και να σώσουμε το αποτέλεσμα με συντομογραφία. Γενική μορφή:

τελεστέος1 op= τελεστέος2

Όπου op είναι οποιοσδήποτε από τους τελεστές +, -, *, %, /, &, ^, |, <<, >>

Παράδειγμα:

a += b

Είναι ισοδύναμο με:

a = a + b

Αντίστοιχα, **a *= b** είναι ισοδύναμο με **a = a * b** και ομοίως για τους άλλους τελεστές

Τελεστές Αύξησης και Μείωσης

Ο **τελεστής αύξησης** ++ μπαίνει πριν ή μετά από το όνομα μίας μεταβλητής και την αυξάνει κατά 1. Ο **τελεστής μείωσης** -- μπαίνει πριν ή μετά από το όνομα μίας μεταβλητής και την μειώνει κατά 1.

Επιθεματικά:

a++ επιστρέφει την τιμή του a πριν την αύξηση

a-- επιστρέφει την τιμή του a πριν την μείωση

Προθεματικά:

++a επιστρέφει την τιμή του a μετά την αύξηση

--a επιστρέφει την τιμή του a μετά την μείωση

Τελεστής Παράθεσης (Comma Operator)

Ο τελεστής παράθεσης υπολογίζει τον πρώτο τελεστέο, στην συνέχεια αγνοεί το αποτέλεσμα και επιστρέφει τον δεύτερο τελεστέο.

Παράδειγμα:

12, 42 επιστρέφει 42

Σχετικά περιορισμένες χρήσεις συνήθως χρησιμοποιείται με τελεστές ανάθεσης.

Πρόγραμμα Υπολογισμού Βαθμολογίας

```
int grade(int final_exam, int homework, int lab){

    return final_exam*50/100+homework*30/100+lab*20/100;
}

int main(int argc, char ** argv) {

    if (argc != 4) {

        printf("Run as: grade final_exam homework lab\n");

        return 1;
    }

    int final_exam = atoi(argv[1]);

    int homework = atoi(argv[2]);

    int lab = atoi(argv[3]);

    int final_grade = grade(final_exam, homework, lab);

    printf("My grade is: %d\n", final_grade);

    return 0;
}
```

Προτεραιότητα (Precedence) & Προσεταιριστικότητα (Associativity)

Η **προτεραιότητα** ενός τελεστή καθορίζει την σειρά με την οποία θα εκτελεστούν οι υπολογισμοί. Για παράδειγμα, ο πολλαπλασιασμός έχει υψηλότερη προτεραιότητα από την πρόσθεση:

$$5 * 6 + 3 * 4 \text{ επιστρέφει ??}$$

Αν δύο τελεστές έχουν την ίδια προτεραιότητα, η **προσεταιριστικότητα** τους καθορίζει την σειρά των υπολογισμών (**αριστερά προς τα δεξιά ή το αντίστροφο**). Για παράδειγμα, ο πολλαπλασιασμός και η διαίρεση έχουν την ίδια προτεραιότητα, ενώ η προσεταιριστικότητα τους είναι από αριστερά προς τα δεξιά

$$90 * 50 / 100 \text{ επιστρέφει ??}$$

Προτεραιότητα (Precedence) & Προσεταιριστικότητα (Associativity)

Η **προτεραιότητα** ενός τελεστή καθορίζει την σειρά με την οποία θα εκτελεστούν οι υπολογισμοί. Για παράδειγμα, ο πολλαπλασιασμός έχει υψηλότερη προτεραιότητα από την πρόσθεση:

$$5 * 6 + 3 * 4 \text{ επιστρέφει ??}$$

Αν δύο τελεστές έχουν την ίδια προτεραιότητα, η **προσεταιριστικότητα** τους καθορίζει την σειρά των υπολογισμών (**αριστερά προς τα δεξιά ή το αντίστροφο**). Για παράδειγμα, ο πολλαπλασιασμός και η διαίρεση έχουν την ίδια προτεραιότητα, ενώ η προσεταιριστικότητα τους είναι από αριστερά προς τα δεξιά

$$90 * 50 / 100 \text{ επιστρέφει ??}$$

Δεν είμαστε σίγουροι για την ακριβή σειρά; Χρησιμοποιούμε παρενθέσεις ()!



Θέση	Τελεστές	Προσεταιριστικότητα
1	() (παρενθέσεις-κλήση συνάρτησης) [] -> . ++ (επιθεματική αύξηση) -- (επιθεματική μείωση)	αριστερά προς δεξιά
2	++ (προθεματική αύξηση) -- (προθεματική μείωση) ! ~ * (έμμεση αναφορά) & (διεύθυνση) + (μοναδιαίο +) - (μοναδιαίο -) (προσαρμογή τύπου) sizeof	δεξιά προς αριστερά
3	* (πολλαπλασιασμός) / %	αριστερά προς δεξιά
4	+ (πρόσθεση) - (αφαίρεση)	αριστερά προς δεξιά
5	<< >>	αριστερά προς δεξιά
6	< <= > >=	αριστερά προς δεξιά
7	== !=	αριστερά προς δεξιά
8	&	αριστερά προς δεξιά
9	^	αριστερά προς δεξιά
10		αριστερά προς δεξιά
11	&&	αριστερά προς δεξιά
12		αριστερά προς δεξιά
13	? :	δεξιά προς αριστερά
14	= += -= *= /= %= &= ^= = <<= >>=	δεξιά προς αριστερά
15	,	αριστερά προς δεξιά

Εκφράσεις και Εντολές (Expressions and Statements)

Κάθε έγκυρος συνδυασμός τελεστών και τελεστέων (μεταβλητών, σταθερών) λέγεται **έκφραση (expression)**. Παράδειγμα έκφρασης:

```
final_exam * 50 / 100 + homework * 30 / 100 + lab * 20 / 100;
```

Μια συνάρτηση αποτελείται από ένα εκφράσεις που συνδυάζονται και εκτελούνται με τον τρόπο που καθορίζουν συντακτικές δομές που λέγονται **εντολές (statements)**. Οι εντολές εκτελούνται διαδοχικά, υπό συνθήκη ή επανειλημμένα, Παράδειγμα:

```
if (argc != 4) {  
    ...  
}
```

Κατηγοριοποίηση Εντολών

1. Κενές Εντολές
2. Εντολές Έκφρασης
3. Σύνθετες Εντολές
4. Εντολές Συνθήκης
5. Εντολές Επανάληψης

Κενή Εντολή (Empty / Null Statement)

Η κενή εντολή δεν εκτελεί τίποτε (no-operation ή no-op) και έχει την μορφή:

```
;; // null statement
```

```
;;;; // 5 null statements
```

Η χρησιμότητά της φαίνεται σε συνδυασμό με άλλες εντολές.

Σημείωση: χρησιμοποιούμε semicolon (;) ως διαχωριστικό εντολών - για να δείξουμε που τελειώνει η εντολή μας.

Εντολή Έκφρασης (Expression Statement)

Εντολή έκφρασης στην C ονομάζουμε μια έκφραση που τελειώνει με semicolon (;).
Παραδείγματα expression statements:

x = 4 ;

y = 7 ;

z = ++y ;

y = z - (x++) ;

z = x - (--y) ;

Ποια η τιμή των x, y, z μετά την εκτέλεση αυτών των εντολών έκφρασης;

Σύνθετη Εντολή (Compound Statement / Block)

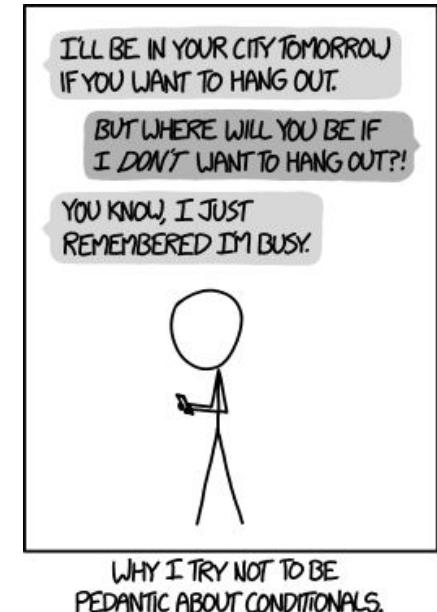
Σύνθεση εντολή ή αλλιώς **block**, λέγεται μια σειρά από εντολές όταν περικλείονται από αγκύλες { }.

Παράδειγμα:

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

Εντολές Ροής Ελέγχου (Control Flow Statements)

Η ροή ελέγχου, δηλαδή η σειρά με την οποία θα εκτελεστούν οι εντολές σε ένα πρόγραμμα καθορίζεται από τις εντολές ροής ελέγχου. Συνήθως οπτικοποιείται με ένα διάγραμμα.



Εντολή if (if Statement)

Η εντολή if εκτελεί μια εντολή αν μια λογική συνθήκη είναι αληθής. Γενική μορφή:

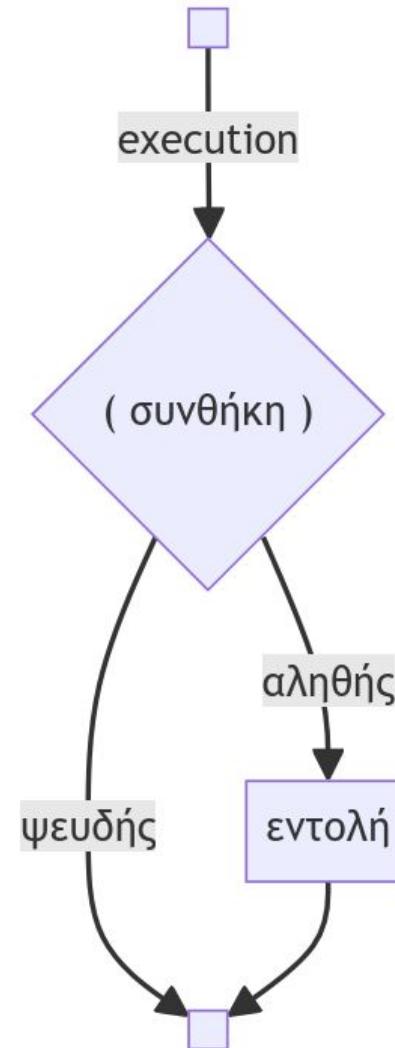
if (συνθήκη)
εντολή

Παραδείγματα:

```
if (year > 1);
```

```
if (year > 1)  
    year++;
```

```
if (year > 1) {  
    year++;  
}
```



Εντολή if (if Statement)

Η εντολή if εκτελεί μια εντολή αν μια **λογική συνθήκη** είναι αληθής. Γενική μορφή:

if (συνθήκη)
εντολή

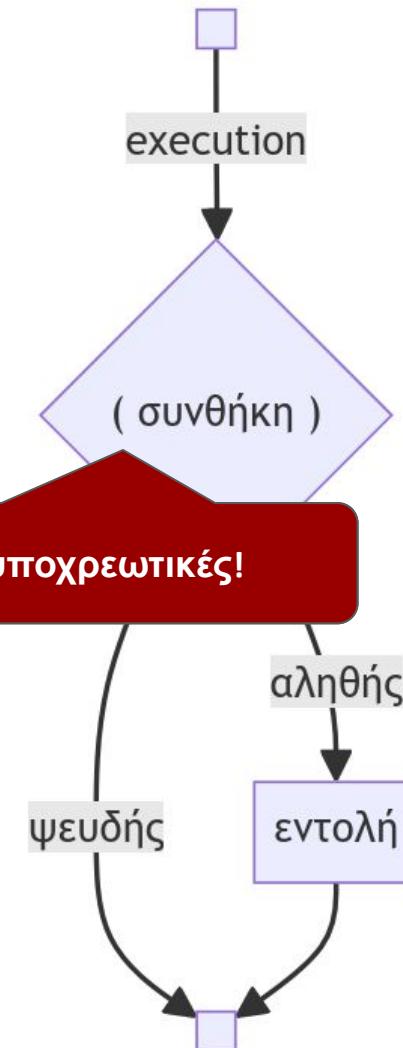
Οι παρενθέσεις είναι υποχρεωτικές!

Παραδείγματα:

if (year > 1)
;

if (year > 1)
year++;

if (year > 1) {
 year++;
}



Εντολή if-else (if-else Statement)

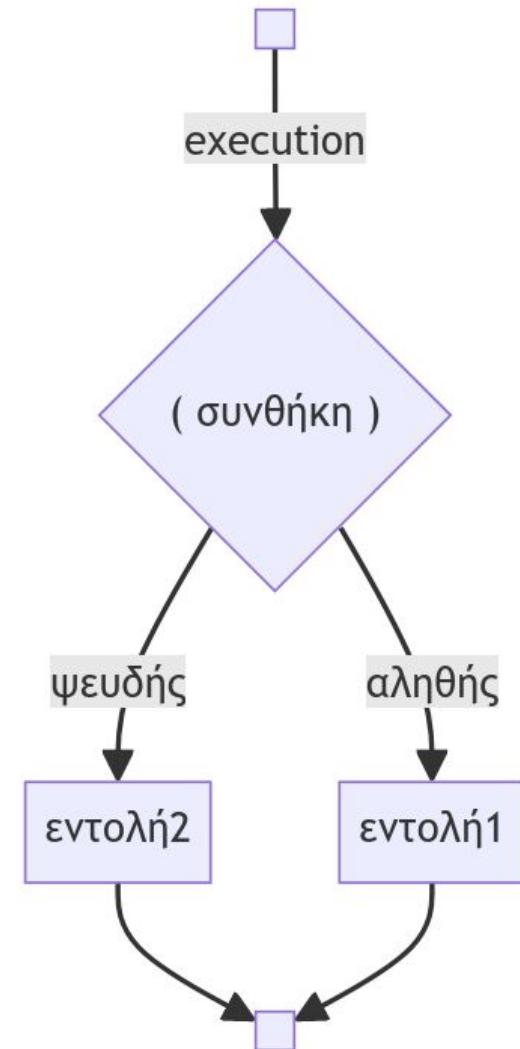
Επέκταση της εντολής if προκειμένου να εκτελέσουμε μια άλλη εντολή αν η λογική συνθήκη είναι ψευδής.

if (συνθήκη)

εντολή1

else

εντολή2



Εντολή if-else (if-else Statement)

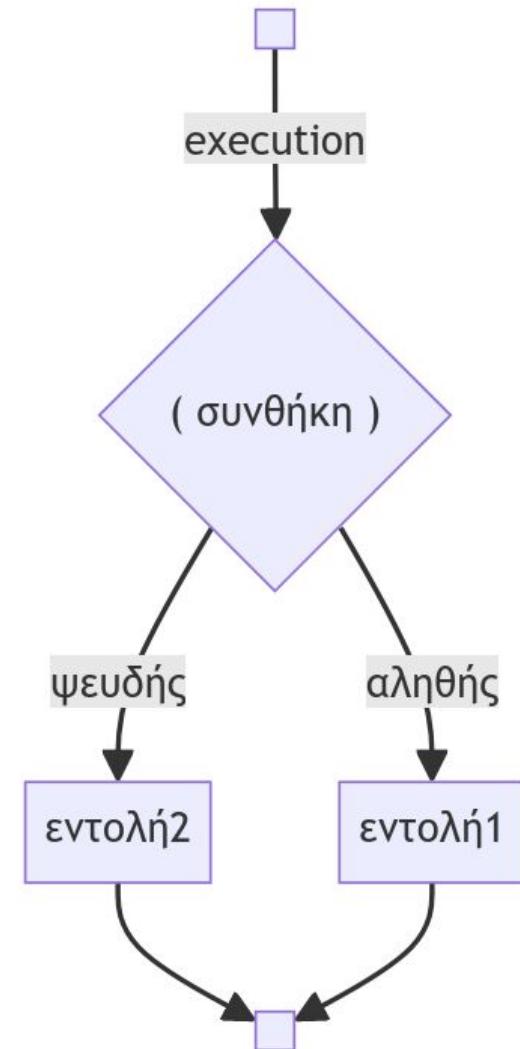
Παραδείγμα:

```
if (x > y)  
    max = x;  
  
else  
  
    max = y;
```

Θα μπορούσαμε να "ζήσουμε" χωρίς else;

Υπάρχει εναλλακτικός τρόπος να γράψουμε το παραπάνω;

Τι κάνουμε όταν έχουμε πάνω από δύο συνθήκες;



Διάλεξη 6 - Εντολές και Ροή Ελέγχου

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Εμφωλευμένες Εντολές if (Nested if)

Πολλές φορές χρειάζεται να ελέγξουμε πάνω από μια συνθήκες και χρησιμοποιούμε εμφωλευμένες if εντολές:

```
lab = -1;  
if (year < 1)  
    { /* empty block */ }  
else if (year == 1)  
    lab = 20;  
else  
    lab = 0;
```

To Dangling else πρόβλημα

Πολλές εμφωλευμένες εντολές if μπορεί να οδηγήσουν σε προβλήματα αμφισημίας (κάτι που δεν μας αρέσει καθόλου στο computer science). Είναι τα δύο προγράμματα ισοδύναμα;

```
lab = -1;  
  
if (year < 1)  
    { /* empty block */ }  
  
else if (year == 1)  
    lab = 20;  
  
else  
    lab = 0;
```

```
lab = -1;  
  
if (year <= 1)  
    if (year == 1)  
        lab = 20;  
  
    else  
        lab = 0;
```

Το πρόβλημα με το Dangling else

Πολλές εμφωλευμένες εντολές if μπορεί να οδηγήσουν σε προβλήματα αμφισημίας (κάτι που δεν μας αρέσει καθόλου στο computer science). Είναι τα δύο προγράμματα ισοδύναμα;

```
lab = -1;  
  
if (year < 1)  
    { /* empty block */ }  
  
else if (year == 1)  
    lab = 20;  
  
else  
    lab = 0;
```

```
lab = -1;  
  
if (year <= 1)  
    if (year == 1)  
        lab = 20;  
  
    else  
        lab = 0;
```

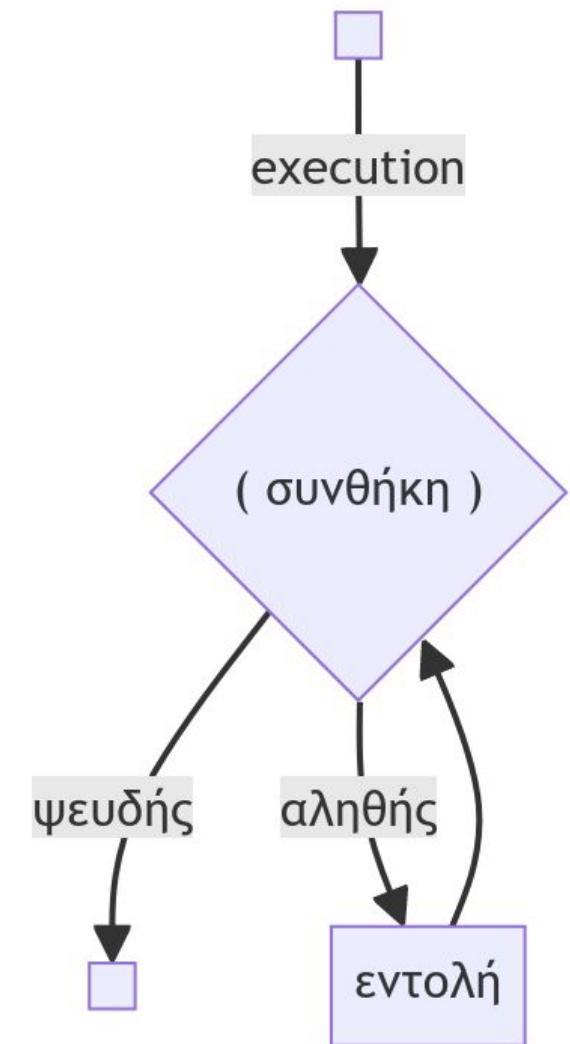
Αν δεν είμαστε σίγουροι χρησιμοποιούμε αγκύλες {} για να υποδείξουμε τα όρια των σύνθετων εντολών

Δομές Επανάληψεις / Loops / Βρόχοι

Εντολή while (while Statement)

Η εντολή while επαναλαμβάνει μια άλλη εντολή όσο η λογική συνθήκη είναι αληθής.

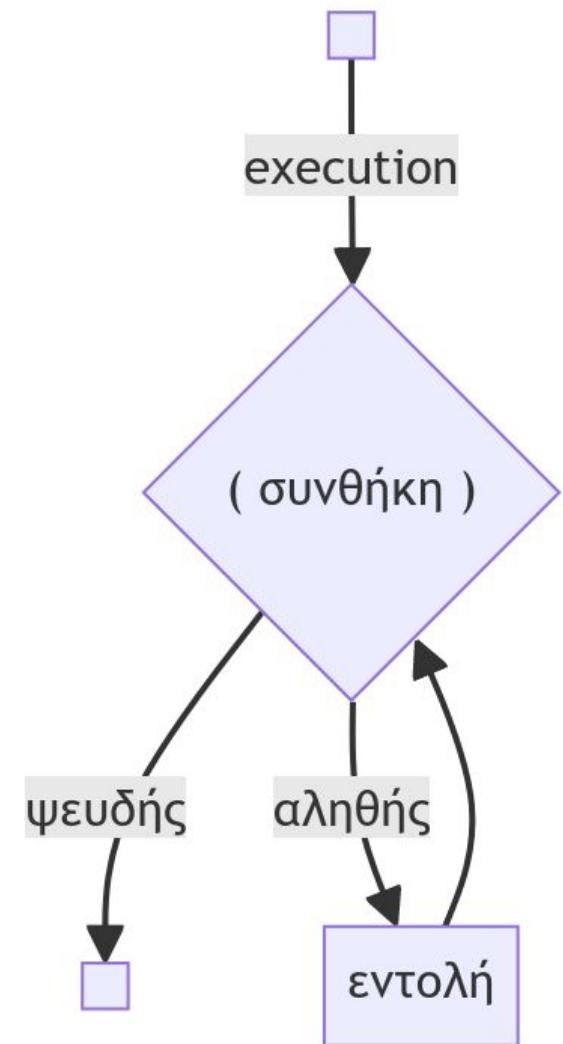
while (συνθήκη)
εντολή



Εντολή while (while Statement)

Παράδειγμα #1: Τι κάνει το παρακάτω πρόγραμμα;

```
int i = 0;  
int sum = 0;  
while (i < 42) {  
    sum += i;  
    i++;  
}  
printf("%d %d\n", i, sum);
```

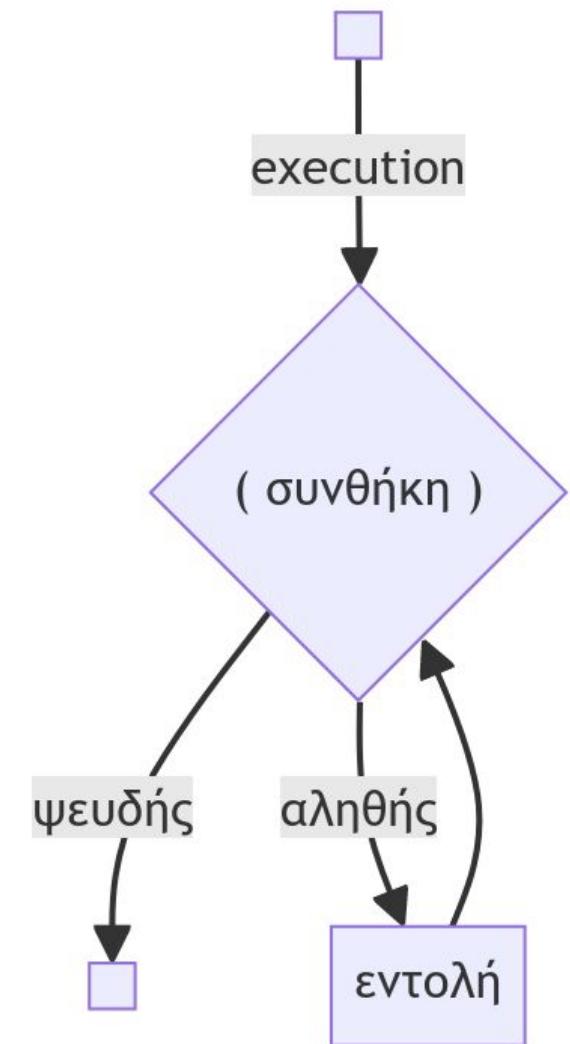


Εντολή while (while Statement)

Παράδειγμα #2: Τι κάνει το παρακάτω πρόγραμμα;

```
while (1);
```

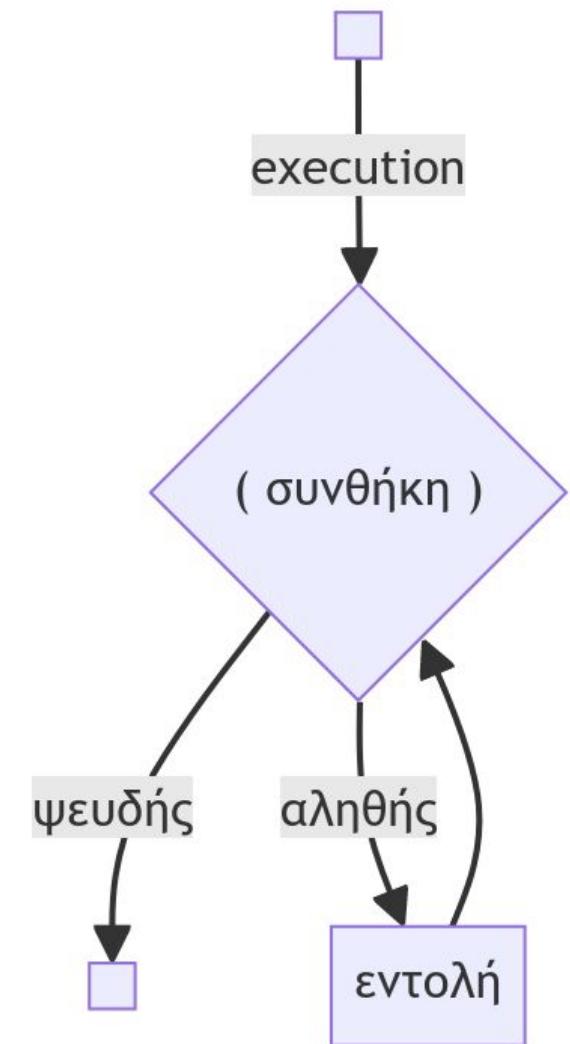
Υπάρχει κάποια χρησιμότητα στο να έχουμε μια λογική συνθήκη που είναι πάντα αληθής;



Εντολή while (while Statement)

Παράδειγμα #3: Τι κάνει το παρακάτω πρόγραμμα;

```
i = 0;  
while (i < 42);  
    printf("%d\n", i++);
```



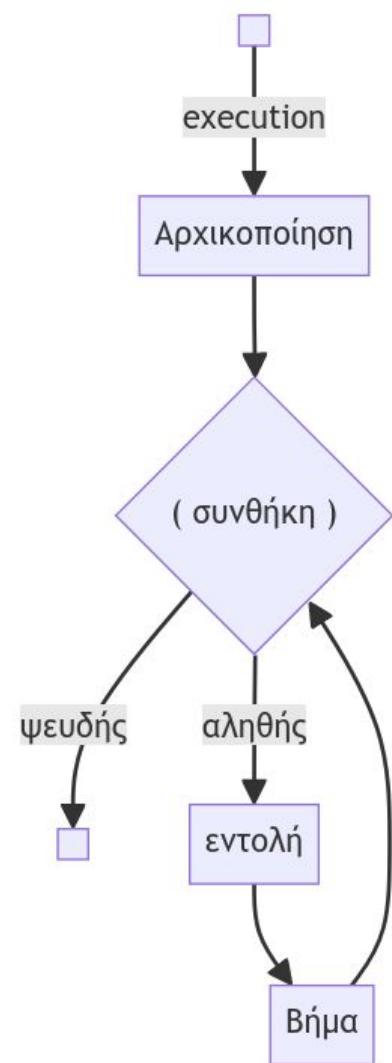
Εντολή for (for Statement)

Η εντολή for: αρχικοποιεί μεταβλητές, επαναλαμβάνει μια εντολή όσο η λογική συνθήκη είναι αληθής και στο τέλος κάθε επανάληψης εκτελεί το βήμα. Γενική μορφή:

for (αρχικοποίηση ; συνθήκη ; βήμα)
εντολή

Παράδειγμα:

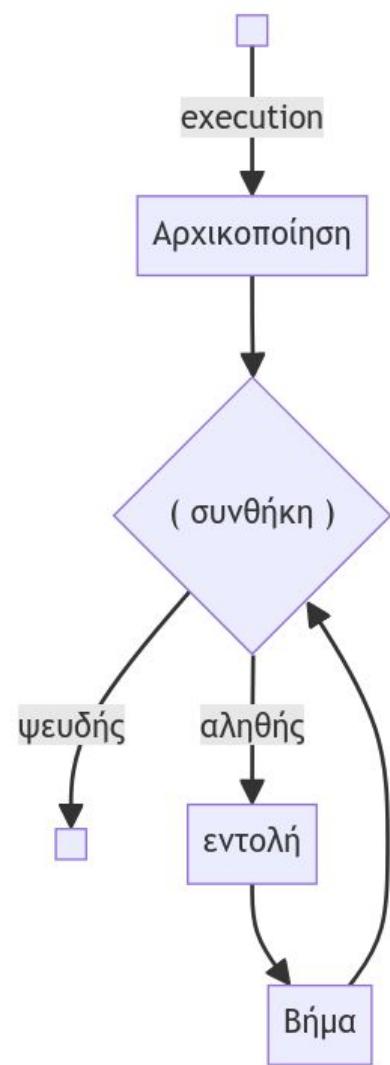
```
int i;  
for ( i = 0 ; i < 100 ; i++ )  
    printf("Hello world\n");
```



Εντολή for (for Statement)

Τι κάνει το παρακάτω πρόγραμμα;

```
for (;;)
```

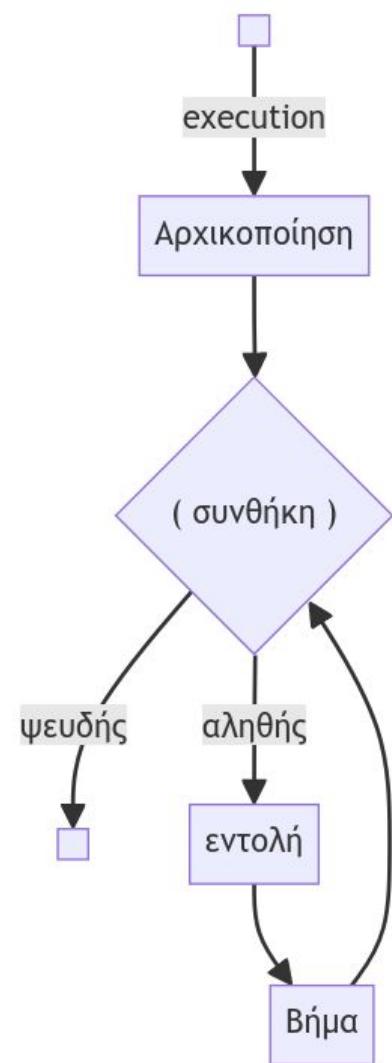


Εντολή for (for Statement)

Γενικό παράδειγμα:

```
int i;  
  
for ( i = 0 ; i < N ; i++ ) {  
    printf("%d\n", i);  
}
```

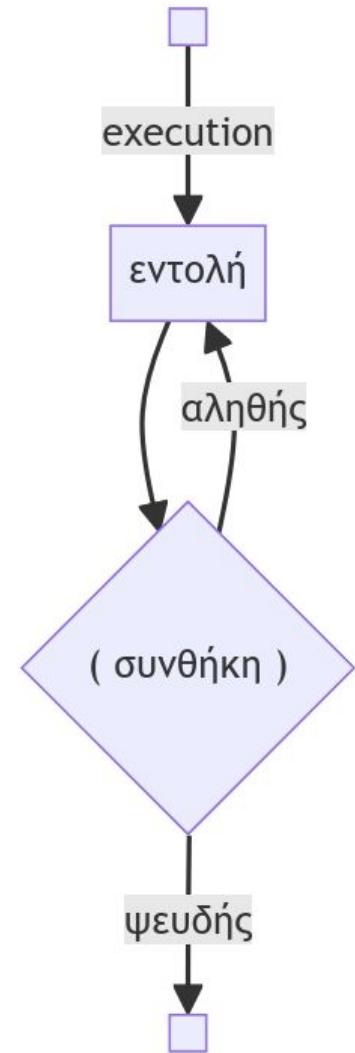
Πόσες φορές θα εκτελεστεί το block εντολών μέσα στην for; Θα τυπωθεί το N; Τι θα συμβεί αν αλλάξω την αρχικοποίηση ή το βήμα;



Εντολή do-while (do-while Statement)

Η εντολή do-while τρέχει πρώτα μια φορά την εντολή και στην συνέχεια ελέγχει την λογική συνθήκη για το αν χρειάζεται άλλη επανάληψη.

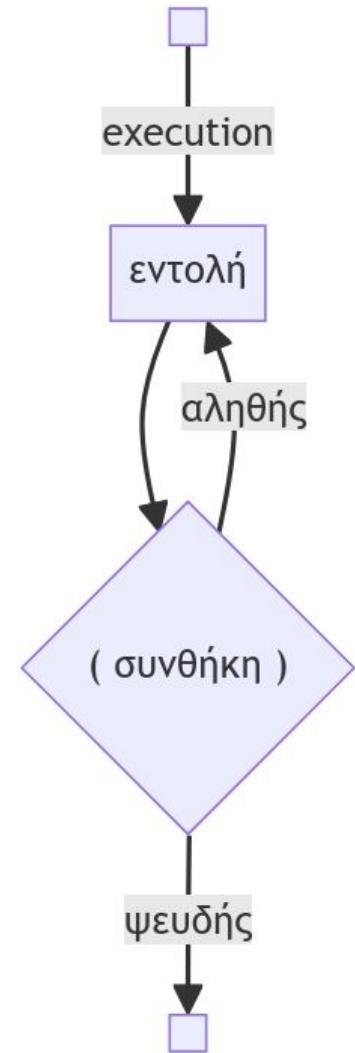
```
do  
    εντολή1  
    while ( συνθήκη );
```



Εντολή do-while (do-while Statement)

Παράδειγμα: Τι κάνει το παρακάτω πρόγραμμα;

```
i = 0;  
do  
    i++;  
    while (i < 42);  
    printf("%d\n", i);
```



Διάλεξη 7 - Επίλυση Προβλημάτων

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Flow

Flow in [positive psychology](#), also known colloquially as being **in the zone** or **locked in**, is the [mental state](#) in which a person performing some activity is fully immersed in a feeling of energized [focus](#), full involvement, and enjoyment in the process of the activity. In essence, flow is characterized by the complete absorption in what one does, and a resulting transformation in one's sense of time. [\[1\]](#)

From [Wikipedia](#)

Θέλω να τυπώσω όλους τους τριψήφιους άρτιους σε φθίνουσα σειρά (998 996 994 ... 100). Πως;

Ένα for loop με μια μεταβλητή που μειώνεται:

```
for(i = 998 ; i >= 100 ; i -= 2) {  
    printf("%d\n", i);  
}
```

Θέλω να το γινόμενο όλων των τριψήφιων περιττών που διαιρούνται με το 7. Πως;

Ένα for loop με μια μεταβλητή που αυξάνεται και έλεγχο για $\text{mod } 7 = 0$:

```
for(product = 1, i = 100 ; i <= 999 ; i++) {  
    if ( i % 7 == 0 )  
        product *= i;  
}
```

Θέλω να το γινόμενο όλων των τριψήφιων περιττών που διαιρούνται με το 7. Πως;

Ένα for loop με μια μεταβλητή που αυξάνεται και έλεγχο για $\text{mod } 7 = 0$:

```
for(product = 1, i = 105 ; i <= 999 ; i += 14) {  
    product *= i;  
}
```

Διάλεξη 8 - Ροή Ελέγχου #2

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Θέλω να βρω το γινόμενο όλων των τριψήφιων περιττών που διαιρούνται με το 7. Πως;

Ένα for loop με μια μεταβλητή που αυξάνεται και έλεγχο για $\text{mod } 7 = 0$:

```
for(product = 1, i = 100 ; i <= 999 ; i++) {
    if ( i % 7 == 0 )
        product *= i;
}
```

Η

```
for(prod = 1, i = 105 ; i <= 999 ; i += 14) {
    prod *= i;
}
```

Θέλω να τυπώσω (αν υπάρχει) τον μικρότερο τριψήφιο που είναι πολλαπλάσιο του 2 και του 5 αλλά όχι του 4. Πως;

```
int found = 0;
for(i = 100 ; i < 1000 && !found; i++) {
    if (i % 2 == 0 && i % 5 == 0 && i % 4 != 0) {
        printf("%d\n", i);
        found = 1;
    }
}
```

Χρησιμοποιούμε μια μεταβλητή `found` για να ελέγξουμε πότε θέλουμε να "σπάσουμε" το loop και να συνεχίσουμε το πρόγραμμά μας. Η C μας προσφέρει και εναλλακτικό τρόπο έκφρασης για να επιτύχουμε το ίδιο αποτέλεσμα.

Εντολή break (break Statement)

Η εντολή **break** μας επιτρέπει να "σπάσουμε" τον τρέχοντα βρόχο τερματίζοντας τον μετά την εκτέλεσή της.

```
while ( ... ) {  
    break;  
}
```



Η ίδια εντολή μας επιτρέπει να "σπάσουμε" και τις άλλες δομές ελέγχου όπως είναι το **do-while** και το **for**.

Παράδειγμα με break

Εύρεση του μικρότερου τριψήφιου που είναι πολλαπλάσιο του 2 και του 5 αλλά όχι του 4 με break:

```
for(i = 100 ; i < 1000; i++) {  
    if (i % 2 == 0 && i % 5 == 0 && i % 4 != 0) {  
        printf("%d\n", i);  
        break;  
    }  
}
```

Εντολή continue (continue Statement)

Η εντολή **continue** μας επιτρέπει να διακόψουμε την τρέχουσα επανάληψη του βρόχου και να συνεχίσουμε στην επόμενη.

```
while ( ... ) {  
    continue;  
    ...  
}
```



Η ίδια εντολή μας επιτρέπει να διακόψουμε την τρέχουσα επανάληψη και στις άλλες δομές ελέγχου όπως είναι το **do-while** και το **for**.

Παράδειγμα με continue

Τύπωσε όλους τους τριψήφιους εκτός από τα πολλαπλάσια του 5:

```
for(i = 100 ; i < 1000; i++) {  
    if (i % 5 == 0) {  
        continue;  
    }  
    printf("%d\n", i);  
}
```

Θέλω να τυπώσω το αγγλικό όνομα κάθε ψηφίου - Πως;

```
if (number == 0)
    printf("zero\n");
else if (number == 1)
    printf("one\n");
else if (number == 2)
    printf("two\n");
...
else if (number == 9)
    printf("nine\n");
else
    printf("unknown\n");
```

Εντολή switch (switch Statement)

Η εντολή ελέγχου **switch** είναι εναλλακτική της **if-else-if** δομής, όταν χρειάζεται να ελέγξουμε μία έκφραση για τις δυνατές τιμές που μπορεί να πάρει και να χειριστούμε την κάθε περίπτωση με διαφορετικό τρόπο.

```
switch (έκφραση) {  
    case σταθερά1:  
    case σταθερά2:  
    ...  
    default:  
}
```

Θέλω να τυπώσω το αγγλικό όνομα κάθε ψηφίου - Πως;

```
switch (number) {  
    case 0:  
        printf("zero\n");  
        break;  
    case 1:  
        printf("one\n");  
        break;  
    ...  
    case 9:  
        printf("nine\n");  
        break;  
    default:  
        printf("unknown\n");  
        break;  
}
```

Θέλω να τυπώσω το αγγλικό όνομα κάθε ψηφίου - Πως;

Τα break
ολοκληρώνουν
την εκτέλεση

```
switch (number) {  
    case 0:  
        printf("zero\n");  
        break;  
    case 1:  
        printf("one\n");  
        break;  
    ...  
    case 9:  
        printf("nine\n");  
        break;  
    default:  
        printf("unknown\n");  
        break;  
}
```

Θέλω να τυπώσω την εποχή κάθε μήνα - Πως;

Μπορούμε να διαχειριστούμε πολλά cases μαζί

```
switch (month) {  
    case 12:  
    case 1:  
    case 2:  
        printf("winter\n");  
        break;  
    case 3:  
    case 4:  
    case 5:  
        printf("spring\n");  
        break;  
    case 6:  
    case 7:  
    case 8:  
        printf("summer\n");  
        break;  
    ...  
}
```

Θέλω να τυπώσω την εποχή κάθε μήνα - Πως;

Σε κάθε case ελέγχεται ισότητα == με την τιμή δεν μπορούμε να γράψουμε άλλες λογικές συνθήκες (πχ >, <, κτλ)

```
switch (month) {  
    case 12:  
    case 1:  
    case 2:  
        printf("winter\n");  
        break;  
    case 3:  
    case 4:  
    case 5:  
        printf("spring\n");  
        break;  
    case 6:  
    case 7:  
    case 8:  
        printf("summer\n");  
        break;  
    ...  
}
```

Κάθε έκφραση που χρησιμοποιούμε στο switch και σταθερά που χρησιμοποιούμε στο case πρέπει να είναι ακέραιος

Εντολή goto (goto Statement)

Η εντολή **goto** μεταφέρει την εκτέλεση του προγράμματος σε άλλη εντολή της ίδιας συνάρτησης με την προϋπόθεση ότι η εντολή έχει μία ετικέτα (label). Γενικής μορφής:

goto label;

Η σύνταξη για την ετικέτα είναι η ακόλουθη:

label:

Παράδειγμα goto

Εύρεση του μικρότερου τριψήφιου που είναι πολλαπλάσιο του 2 και του 5 αλλά όχι του 4 με goto:

```
for(i = 100 ; i < 1000; i++) {  
  
    if (i % 2 == 0 && i % 5 == 0 && i % 4 != 0) {  
        printf("%d\n", i);  
        goto exit_for;  
    }  
}  
exit_for:
```

Δομημένος προγραμματισμός σημαίνει OXI goto

Η χρήση goto συχνά οδηγεί σε δυσνόητο κώδικα (χρήσιμο για [obfuscation contests](#) και [spaghetti code!](#)) και καλό είναι να αποφεύγεται. Μπορεί κανείς να γράψει εκατομμύρια γραμμές κώδικα χωρίς να την χρειαστεί.

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

dynamic progress is only characterized by the call of the procedure we refer to. we can characterize the progress by the textual indices, the length of the dynamic depth of procedure calls.

Να χρησιμοποιήσω goto?

'ΟΧΙ

I COULD RESTRUCTURE
THE PROGRAM'S FLOW
OR USE ONE LITTLE
'GOTO' INSTEAD.



EH, SCREW GOOD PRACTICE.
HOW BAD CAN IT BE?

goto main_sub3;

COMPILE



Διάλεξη 9 - Δεδομένα Εισόδου

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Τι θα τυπώσει το παρακάτω πρόγραμμα;

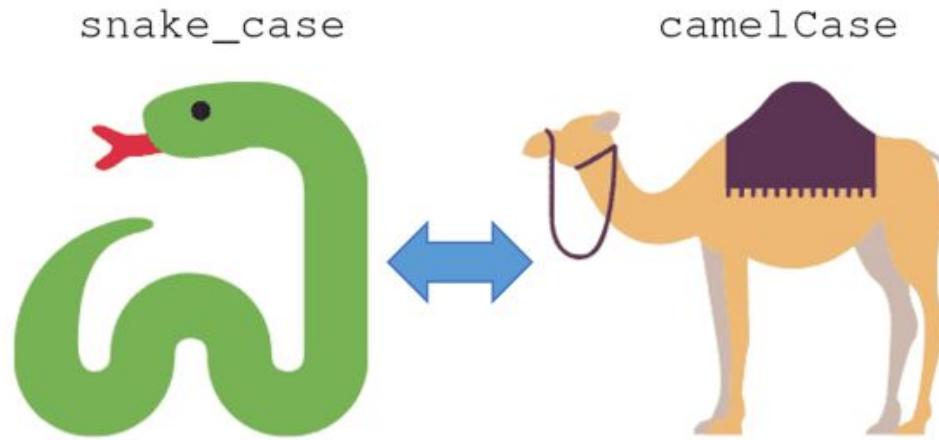
```
#include <stdio.h>
int main() {
    float a = 3.1;

    if(a == 3.1)
        printf("OK\n");
    else
        printf("Not OK\n");
    return 0;
}
```

\$./one
Not OK

Υπάρχει κάποια εξήγηση;

camelCase vs snake_case



Για κώδικα C/Python συνιστώ snake_case

Παράδειγμα: `pi_approx` είναι καλύτερο (υποκειμενικό) από `piApprox`

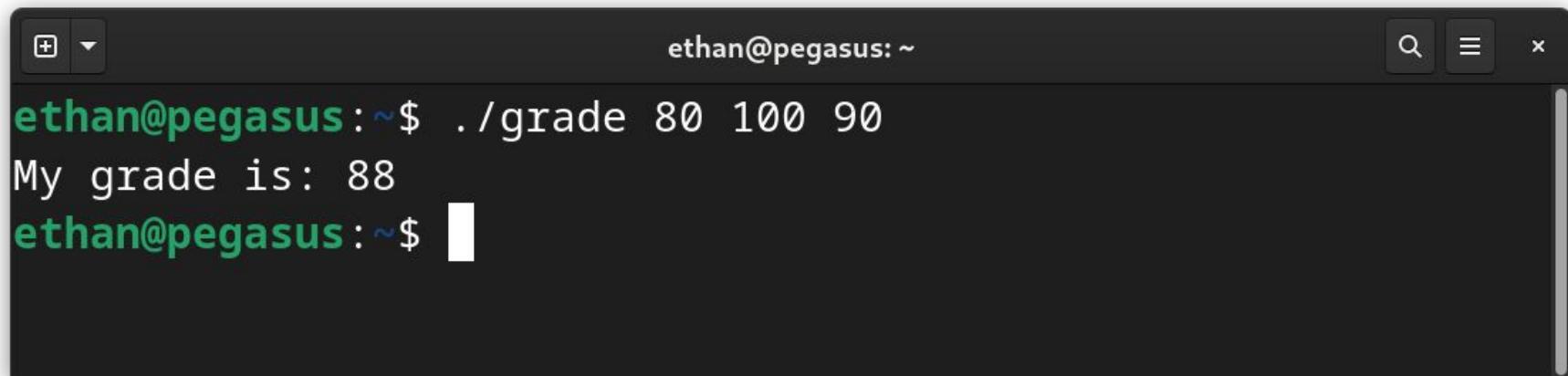
Δεδομένα Εισόδου και Εξόδου (Input and Output Data)



Δεδομένα Εισόδου (Input Data)

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

1. **Ορίσματα** στην γραμμή εντολών



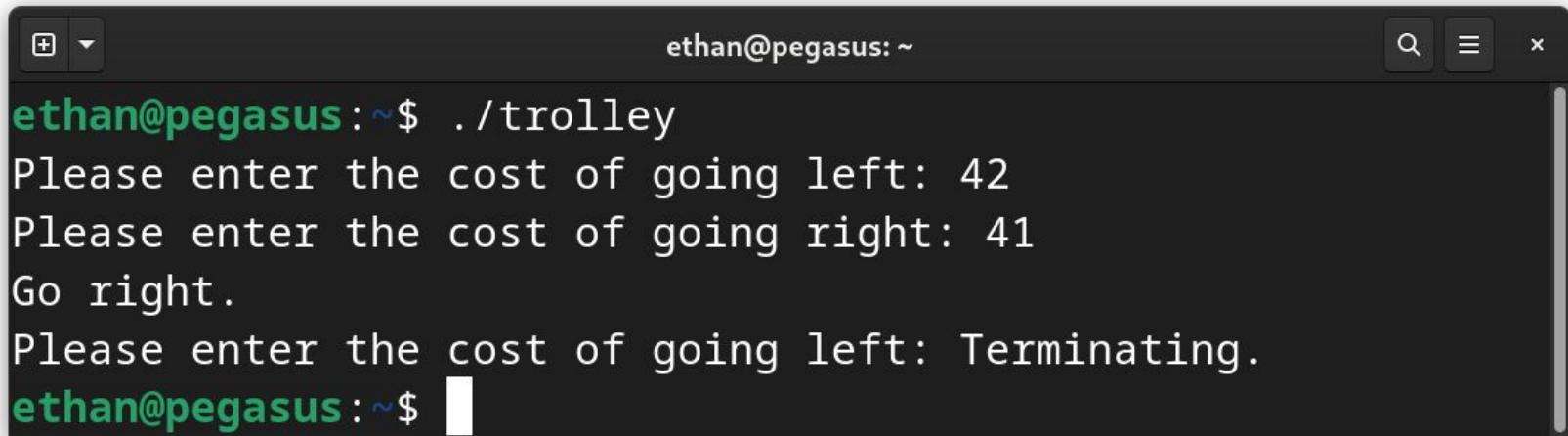
A screenshot of a terminal window titled "ethan@pegasus: ~". The window contains the following text:

```
ethan@pegasus:~$ ./grade 80 100 90
My grade is: 88
ethan@pegasus:~$
```

Δεδομένα Εισόδου (Input Data) - 2/4

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

2. Γράφοντας κείμενο στην **πρότυπη είσοδο** (standard input ή `stdin`) συνήθως με το πληκτρολόγιο

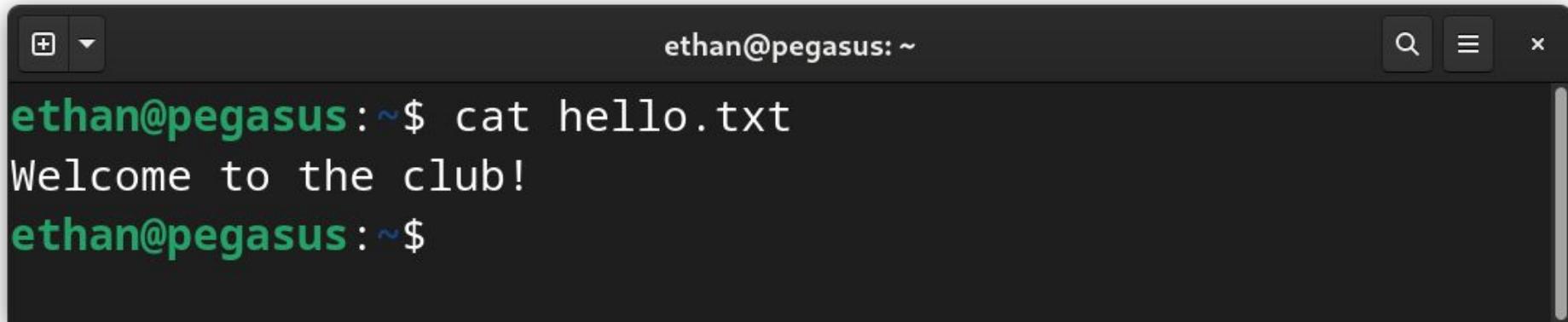


```
ethan@pegasus: ~$ ./trolley
Please enter the cost of going left: 42
Please enter the cost of going right: 41
Go right.
Please enter the cost of going left: Terminating.
ethan@pegasus: ~$
```

Δεδομένα Εισόδου (Input Data) - 3/4

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

3. Διαβάζοντας **αρχεία** από το σύστημα αρχείων (επόμενες διαλέξεις)

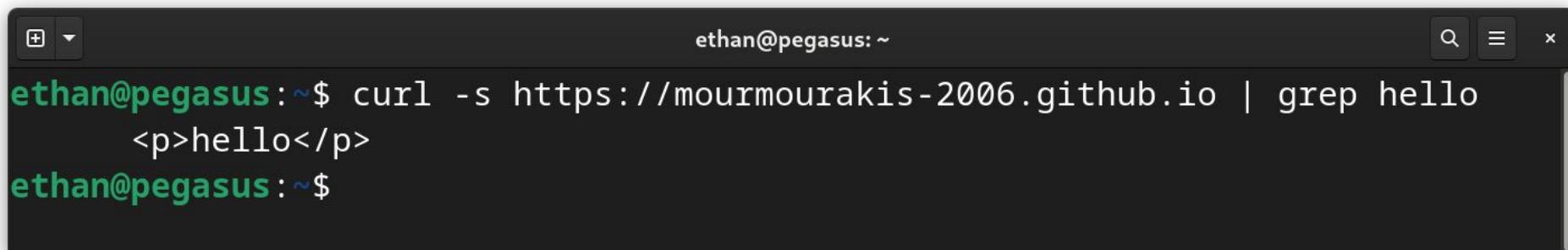


```
ethan@pegasus: ~$ cat hello.txt
Welcome to the club!
ethan@pegasus: ~$
```

Δεδομένα Εισόδου (Input Data) - 4/4

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

4. Διαβάζοντας από το **δίκτυο** ή άλλες πηγές - π.χ., User Interface (σε επόμενα εξάμηνα)



A screenshot of a terminal window titled "ethan@pegasus: ~". The window contains the following command and its output:

```
ethan@pegasus:~$ curl -s https://mourmourakis-2006.github.io | grep hello
<p>hello</p>
ethan@pegasus:~$
```

Δεδομένα Εισόδου (Input Data)

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

1. **Ορίσματα** στην γραμμή εντολών
2. Γράφοντας κείμενο στην **πρότυπη είσοδο** (standard input ή `stdin`)
3. Διαβάζοντας **αρχεία** από το σύστημα αρχείων (επόμενες διαλέξεις)
4. Διαβάζοντας από το **δίκτυο** ή άλλες πηγές (άλλα εξάμηνα)

50%



Η συνάρτηση `getchar()`

Η συνάρτηση `getchar()` ορίζεται στο `stdio.h`, διαβάζει έναν χαρακτήρα εάν υπάρχει από το `stdin` του προγράμματος και τον επιστρέφει ως ακέραιο. Αν δεν υπάρχει, επιστρέφει την τιμή End-Of-File / EOF (-1).

Η συνάρτηση `getchar()` είναι "έτοιμη" για χρήση από μας από το `stdio.h` - πως μπορώ να βρω πως συμπεριφέρεται;

Ανοίγω ένα τερματικό και τρέχω την
`getchar()`!

Η συνάρτηση `getchar()`

Η συνάρτηση `getchar()` ορίζεται στο `stdio.h`, διαβάζει έναν χαρακτήρα εάν υπάρχει από το `stdin` του προγράμματος και τον επιστρέφει ως ακέραιο. Αν δεν υπάρχει, επιστρέφει την τιμή End-Of-File / EOF (-1). Η συνάρτηση έχει την ακόλουθη μορφή:

```
int getchar();
```

Δεν παίρνει κανένα όρισμα και επιστρέφει έναν ακέραιο.

Χρήση της συνάρτησης getchar()

Για να διαβάσουμε έναν χαρακτήρα και να τον τυπώσουμε, γράφουμε:

```
#include <stdio.h>

int main() {
    printf("Gimme a char: ");
    int ch = getchar();
    if (ch != EOF) {
        printf("You gave the char: %c\n", ch);
    } else {
        printf("input ended\n");
    }
    return 0;
}
```

Χρήση της συνάρτησης getchar()

Για να διαβάσουμε έναν χαρακτήρα και να τον τυπώσουμε, γράφουμε:

```
#include <stdio.h>

int main() {

    printf("Gimme a char: ");
    int ch = getchar();
    if (ch != EOF) {
        printf("You gave the char: %c\n", ch);
    } else {
        printf("input ended\n");
    }
    return 0;
}
```

Η getchar θα διαβάσει μόνο έναν χαρακτήρα

```
$ ./chartest
Gimme a char: BBB
You gave the char: B
```

Χρήση της συνάρτησης getchar()

Συνήθως, πρέπει να περιμένουμε μέχρι να πατήσουμε **Enter** προκειμένου οι χαρακτήρες που πληκτρολογήσαμε να φτάσουν το πρόγραμμα.



Program

Τα δεδομένα "αποθηκεύονται" προσωρινά σε έναν Buffer μέχρι να σταλεί καινούρια γραμμή και να προωθηθούν στο πρόγραμμα

Χρήση της συνάρτησης getchar()

Για να δείξουμε ότι τελείωσαν τα δεδομένα εισόδου, στο Linux συνήθως πρέπει να πατήσουμε **Ctrl+D** (EOF)



Program

Όταν πατήσουμε **Ctrl+D** όλα τα δεδομένα που βρίσκονται στον buffer θα προωθηθούν στο πρόγραμμα (και χωρίς καινούρια γραμμή)

Χρήση της συνάρτησης getchar()

Διαδοχικές κλήσεις της getchar() διαβάζουν διαδοχικούς χαρακτήρες. Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int ch, sum = 0;
    printf("Enter characters: ");
    while( (ch = getchar()) != '\n' && ch != EOF ) {
        printf("%c", ch);
        sum++;
    }
    printf("\nTotal characters: %d\n", sum);
    return 0;
}
```

```
$ ./charcount
Enter characters: we'll always have paris
we'll always have paris
Total characters: 23
```

Η συνάρτηση putchar()

Η συνάρτηση **putchar()** ορίζεται στο stdio.h, παίρνει έναν χαρακτήρα ως όρισμα, τον τυπώνει στο **stdout** του προγράμματος και τον επιστρέφει ως ακέραιο. Αν κάτι δεν πάει καλά στο τύπωμα, επιστρέφει την τιμή EOF (-1). Η συνάρτηση έχει την ακόλουθη μορφή:

```
int putchar(int c);
```

Προκειμένου να τυπώσουμε έναν χαρακτήρα 'C' απλά γράφουμε `putchar('C');`. Η συνάρτηση αυτή είναι είναι ένα απλούστερο υποσύνολο της `printf`.

Τι πρόβλημα έχει η παρακάτω υλοποίηση της cat

```
#include <stdio.h>

int main() {
    char c;
    while((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

```
$ echo -e "hello\xffworld" | ./cat
hello
$ echo -e "hello\xffworld" | cat
hello♦world
```

Προσοχή: πάντα αναθέτουμε την τιμή επιστροφής της getchar() εκτός και αν είμαστε σίγουροι για το τι κάνουμε

Θέλω να διαβάσω δύο αριθμούς από την πρότυπη είσοδο και να τους προσθέσω. Πως;

\$./addnums

Give me a number: 40

Give me another number: 2

Total: 42

```
#define ERROR -1

int getinteger(int base) {

    int ch;
    int val = 0;

    while ((ch = getchar()) != '\n')

        if (ch >= '0' && ch <= '0' + base - 1)

            val = base * val + (ch - '0');

        else

            return ERROR;

    return val;

}
```

Κάνοντας χρήση της getchar και φτιάχνοντας μια συνάρτηση getinteger προκειμένου να διαβάσουμε τους χαρακτήρες έναν-έναν και να τους μετατρέψουμε σε αριθμό. Υπάρχει άλλος τρόπος να επιτύχουμε το ίδιο αποτέλεσμα;

Η συνάρτηση `scanf`

Η συνάρτηση `scanf` ορίζεται στο header file `stdio.h` και χρησιμοποιείται για να διαβάζει δεδομένα εισόδου πολλών τύπων από το `stdin` του προγράμματος και να αποθηκεύσει τις τιμές τους σε μεταβλητές. Αν επιτύχει, επιστρέφει πόσα δεδομένα εισόδου διάβασε. Αν αποτύχει, επιστρέφει την τιμή End-Of-File / EOF (-1).

Πως μπορώ να βρω πως συμπεριφέρεται;

Ανοίγω ένα τερματικό και τρέχω
`man scanf`!

Η συνάρτηση `scanf`

Η συνάρτηση `scanf` ορίζεται στο header file `stdio.h` και χρησιμοποιείται για να διαβάζει δεδομένα εισόδου πολλών τύπων από το `stdin` του προγράμματος και να αποθηκεύσει τις τιμές τους σε μεταβλητές. Αν επιτύχει, επιστρέφει πόσα δεδομένα εισόδου διάβασε. Αν αποτύχει, επιστρέφει την τιμή `End-Of-File / EOF` (-1). Η συνάρτηση έχει την ακόλουθη μορφή:

```
int scanf(const char *restrict format, ...);
```

Έχει μια συμβολοσειρά μορφοποίησης
(format string)

Δέχεται όσα ορίσματα περάσουμε
(άλλο μάθημα)

Η συνάρτηση `scanf`

Η συνάρτηση `scanf` ορίζεται στο header file `stdio.h` και χρησιμοποιείται για να διαβάζει δεδομένα εισόδου πολλών τύπων από το `stdin` του προγράμματος και να αποθηκεύσει τις τιμές τους σε μεταβλητές. Αν επιτύχει, επιστρέφει πόσα δεδομένα εισόδου διάβασε. Αν αποτύχει, επιστρέφει την τιμή End-Of-File / EOF (-1). Η συνάρτηση έχει την ακόλουθη μορφή:

```
int scanf(const char *restrict format, ...);  
int printf(const char *restrict format, ...);
```

Είναι η συμμετρική της `printf` για διάβασμα αντί για εκτύπωση

Χρήση της συνάρτησης scanf

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int n;

    printf("Gimme a number: ");
    scanf("%d", &n);
    printf("Square: %d\n", n * n);

    return 0;
}
```

Τι θα γινόταν αν γράφαμε scanf("%d" , n); Γιατί;
\$./scanf
Gimme a number: 3
Segmentation fault

Περνάμε την **διεύθυνση (spoiler)** της μεταβλητής n στην μνήμη ώστε η scanf να μπορέσει να αναθέσει την τιμή που διάβασε

Τυπώνει στο stdout το τετράγωνο του αριθμού που γράψαμε στο stdin

Χρήση της συνάρτησης scanf

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int n;

    printf("Gimme a number: ");
    scanf("%d", &n);
    printf("Square: %d\n", n * n);

    return 0;
}
```

Είναι σωστό αυτό το πρόγραμμα;

Όχι καθώς δεν ελέγχουμε την τιμή επιστροφής της scanf (EOF ή ίσως 0!)

Χρήση της συνάρτησης scanf

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int n;
    printf("Gimme a number: ");
    scanf("%d", &n);
    printf("Square: %d\n", n * n);
    return 0;
}
```

```
$ ./scanf
Gimme a number: 16
Square: 256
$ ./scanf
Gimme a number: Square:
1068701481
$ ./scanf
Gimme a number: hello
Square: 1072038564
```

Χρήση της συνάρτησης scanf - Πολλά ορίσματα

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int n1, n2;
    printf("Gimme two numbers: ");
    scanf("%d %d", &n1, &n2);
    printf("Result: %d\n", n1 * n2);
    return 0;
}
```

```
$ ./scanf2
Gimme two numbers: 2 4
Result: 8
```

Καθώς ψάχνει για δεκαδικό ψηφίο, η scanf αγνοεί τους κενούς χαρακτήρες ή αλλαγές γραμμής

Τι κάνει το παρακάτω πρόγραμμα;

```
#define ERROR -1                                // Return value for illegal character

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

Τι κάνει το παρακάτω πρόγραμμα;

```
int i, ch, total = 0;  
  
int letfr[26]; // Letter occurrences and frequencies array  
  
for (i=0 ; i < 26 ; i++)  
    letfr[i] = 0;  
  
while ((ch = getchar()) != EOF) {  
  
    if (ch >= 'A' && ch <= 'Z') {  
  
        letfr[ch-'A']++; // Found upper case letter  
  
        total++;  
    }  
  
    if (ch >= 'a' && ch <= 'z') {  
  
        letfr[ch-'a']++; // Found lower case letter  
  
        total++;  
    }  
}
```

Διάλεξη 10 - Πίνακες

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

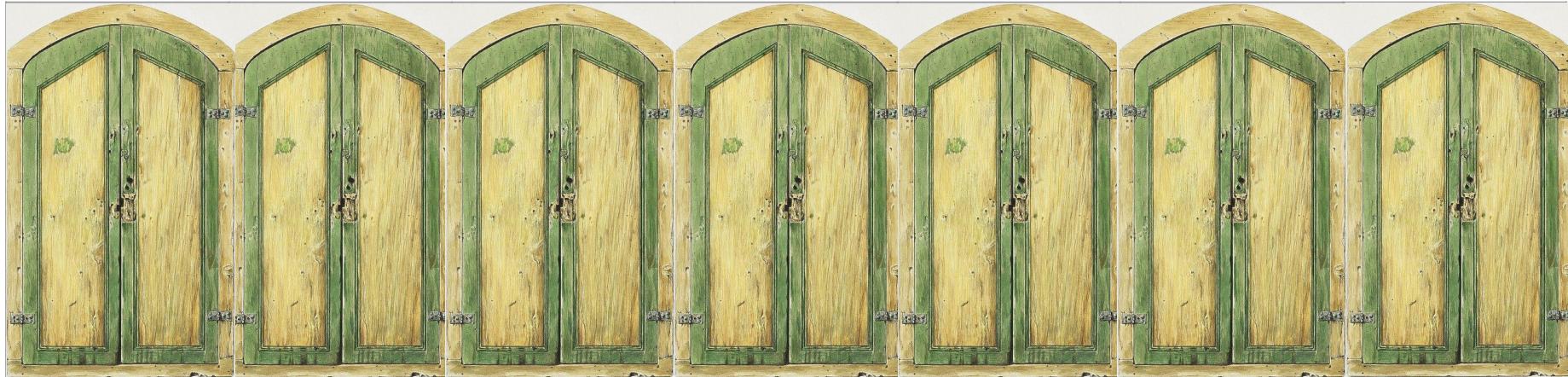
Πίνακες

Έστω ότι έχω 100 αρκουδάκια και
ψάχνω να βρω το μεγαλύτερο. Τι
μπορώ να κάνω για να το βρω;





Στόχος: ελαχιστοποίηση του κόπου (efficient aka τεμπέλης)



Πως θα κωδικοποιήσουμε αυτό το πρόβλημα σε C;

Έστω ότι χρησιμοποιούμε έναν ακέραιο (int) για να αναπαραστήσουμε το κάθε αρκουδάκι. Ας γράψουμε τον κώδικα:

```
int bear0, bear1, bear2, bear3, bear4, bear5, bear6, bear7, bear8, bear9, bear10, bear11, bear12, bear13,  
bear14, bear15, bear16, bear17, bear18, bear19, bear20, bear21, bear22, bear23, bear24, bear25, bear26,  
bear27, bear28, bear29, bear30, bear31, bear32, bear33, bear34, bear35, bear36, bear37, bear38, bear39,  
bear40, bear41, bear42, bear43, bear44, bear45, bear46, bear47, bear48, bear49, bear50, bear51, bear52,  
bear53, bear54, bear55, bear56, bear57, bear58, bear59, bear60, bear61, bear62, bear63, bear64, bear65,  
bear66, bear67, bear68, bear69, bear70, bear71, bear72, bear73, bear74, bear75, bear76, bear77, bear78,  
bear79, bear80, bear81, bear82, bear83, bear84, bear85, bear86, bear87, bear88, bear89, bear90, bear91,  
bear92, bear93, bear94, bear95, bear96, bear97, bear98, bear99;
```

Πως θα κωδικοποιήσουμε αυτό το πρόβλημα σε C;

Έστω ότι χρησιμοποιούμε έναν ακέραιο (int) για να αναπαραστήσουμε το κάθε αρκουδάκι. Ας γράψουμε τον κώδικα:

```
int bear0, bear1, bear2, bear3, bear4, bear5, ..., bear99, max;  
  
bear0 = 42; bear1 = 4; bear2 = 2; ... // αρχικοποίηση μεταβλητών  
  
// find the max here:
```

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE
EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS	
	1 HOUR	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS	
	6 HOURS			2 MONTHS	2 WEEKS	1 DAY	
	1 DAY				8 WEEKS	5 DAYS	

Πολύ επαναληπτικό -
μπορούμε να γλυτώσουμε
χρόνο;

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειρίζομαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

τύπος όνομα[μέγεθος];

Ο **τύπος (type)** της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για κάθε στοιχείο του πίνακα

Το **όνομα (name)** του πίνακα κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης θα τον αποθηκεύσει

Το **μέγεθος (size)** του πίνακα λέει στον μεταγλωττιστή πόσες θέσεις αυτού του τύπου να κρατήσει - στατικό: αφού δηλωθεί δεν αλλάζει κατά την εκτέλεση

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειρίζομαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Ο **τύπος (type)** της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για κάθε στοιχείο του πίνακα

Το **όνομα (name)** του πίνακα κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης θα τον αποθηκεύσει

Το **μέγεθος (size)** του πίνακα λέει στον μεταγλωττιστή πόσες θέσεις αυτού του τύπου να κρατήσει - στατικό: αφού δηλωθεί δεν αλλάζει κατά την εκτέλεση

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **Θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Bytes 0-3

Bytes 4-7

Bytes 8-11

Bytes 12-15

Bytes 400-403

Μνήμη

b0	b1	b2	b3
b4	b5	b6	b7
b8	b9	b10	b11
b12	b13	b14	b15
...			
b400	b401	b402	b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

bears[0]	Μνήμη			
Bytes 0-3	b0 b1 b2 b3			
Bytes 4-7	b4	b5	b6	b7
Bytes 8-11	b8	b9	b10	b11
Bytes 12-15	b12	b13	b14	b15
...				
Bytes 400-403	b400	b401	b402	b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Μνήμη			
bears[1]	b0	b1	b2
Bytes 4-7	b4	b5	b6
Bytes 8-11	b8	b9	b10
Bytes 12-15	b12	b13	b14
	...		
Bytes 400-403	b400	b401	b402
			b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Μνήμη	b0	b1	b2	b3
Bytes 0-3	b4	b5	b6	b7
Bytes 4-7	b8	b9	b10	b11
Bytes 8-11	b12	b13	b14	b15
bears[99]	...			
Bytes 400-403	b400	b401	b402	b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειρίζομαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση πίνακα γίνεται σε μορφή:

```
int bears[100];
```

Ο πίνακας bears
καταλαμβάνει $4 * 100 = 400$ bytes
μνήμης στις διευθύνσεις 4-403

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Bytes 400-403

Μνήμη			
b0	b1	b2	b3
b4	b5	b6	b7
b8	b9	b10	b11
b12	b13	b14	b15
...			
b400	b401	b402	b403

Χρήση Στοιχείων Πίνακα

Ένας πίνακας N στοιχείων, έχει στοιχεία με θέσεις από το 0 μέχρι το $N-1$

Κάθε στοιχείο του πίνακα μπορεί να χρησιμοποιηθεί όπως μια μεταβλητή του ίδιου τύπου σε εκφράσεις ανάθεσεις, τελεστές και συνθήκες.

```
bears[4] = 42;
```

```
bears[8] = bears[2] + 42;
```

```
bears[ bears[4] ] = 2
```

Αρχικοποίηση Πίνακα

Παρεμφερής με την σύνταξη για αρχικοποίηση μεταβλητής:

```
int bears[100] = {  
    11, 25, 26, 31, 14, 13, 19, 3, 2, 19, 30, 7, 28, 9, 20, 19,  
    19, 1, 23, 15, 21, 18, 0, 25, 26, 20, 30, 29, 15, 29, 24, 9,  
    5, 20, 27, 13, 26, 14, 10, 27, 10, 3, 18, 31, 11, 19, 15, 9,  
    20, 15, 13, 31, 15, 9, 22, 22, 17, 30, 25, 14, 18, 0, 22, 13,  
    17, 2, 26, 10, 0, 9, 11, 10, 24, 2, 25, 18, 26, 31, 1, 18, 31,  
    1, 31, 9, 20, 15, 28, 17, 20, 14, 28, 11, 20, 14, 27, 11, 13,  
    6, 26, 31  
}
```

Εύρεση Μέγιστου Στοιχείου σε Πίνακα

Θέλουμε μια συνάρτηση `find_max` που να παίρνει έναν πίνακα 100 στοιχείων και να γυρίζει το μέγιστο:

Εύρεση Μέγιστου Στοιχείου σε Πίνακα

Θέλουμε μια συνάρτηση `find_max` που να παίρνει έναν πίνακα 100 στοιχείων και να γυρίζει το μέγιστο:

```
int find_max(int bears[100]) {  
    int i, max = bears[0];  
    for(i = 1; i < 100; i++) {  
        if (bears[i] > max) max = bears[i];  
    }  
    return max;  
}
```

Εντοπισμός Θέσεων Μνήμης Στοιχείου Πίνακα

Σε ποια διεύθυνση μνήμης βρίσκεται το στοιχείο (τύπου int) bears[2];

$$\text{Αρχή του πίνακα} + 2 * \text{sizeof(int)} = 4 + 2 * 4 = 12$$

Μνήμη

Bytes 0-3	b0	b1	b2	b3
Bytes 4-7	b4	b5	b6	b7
Bytes 8-11	b8	b9	b10	b11
Bytes 12-15	b12	b13	b14	b15
...				
Bytes 400-403	b400	b401	b402	b403

Το στοιχείο
bears[2] ξεκινάει
από το 12ο byte
της μνήμης

Δήλωση Πινάκων Διαφορετικών Τύπων

Πίνακες μπορούν να οριστούν για όλους τους τύπους της C. Παράδειγμα:

```
int a[1024];
```

```
char b[2048];
```

```
double c[512];
```

Ποιος από τους παραπάνω πίνακες καταλαμβάνει περισσότερη μνήμη;

Πίνακας Χαρακτήρων (String)

Ένας πίνακας από χαρακτήρες λέγεται και **αλφαριθμητικό / συμβολοσειρά** (string). Λόγω της συχνής χρήσης τους, έχουμε αρκετές συντομεύσεις για αυτούς (θα δούμε και σε επόμενα μαθήματα). Οι τρεις παρακάτω δηλώσεις είναι ισοδύναμες:

```
char hello[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\n', '\0'};  
char hello[] = {72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 10, 0};  
char hello[] = "Hello World\n";
```

Προσοχή: τα string
τερματίζονται πάντα με
το null byte

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\n'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------

Μπορώ να αναθέσω τα στοιχεία ενός πίνακα σε άλλον;

```
int a[3] = {1, 2, 3};
```

```
int b[3] = {4, 5, 6};
```

```
b = a;
```



Δεν επιτρέπεται στην C!

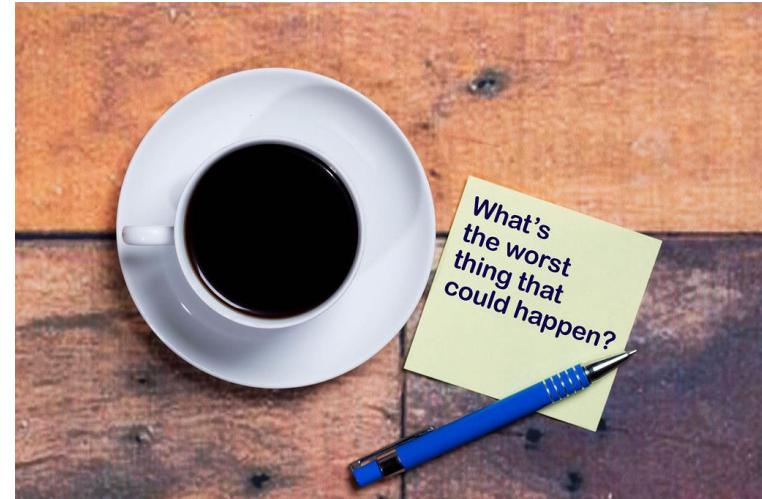
Που είναι χρήσιμοι οι πίνακες;

Είναι η βασικότερη **δομή δεδομένων (data structure)** στην πλειοψηφία των γλωσσών προγραμματισμού

- Μοντελοποίηση συνόλου τιμών ίδιου τύπου
- Μαζική δέσμευση και χρήση μνήμης με μια δήλωση
- Άμεση αποθήκευση, προσπέλαση και μετατροπή δεδομένων

Τι θα συμβεί αν προσπελάσω εκτός ορίων πίνακα;

- Υπερχείλιση (overflow) - π.χ. bears[100]
- Υποχείλιση (underflow) - π.χ. bears[-1]
- Το standard της γλώσσας κατηγοριοποιεί αυτήν την χρήση ως "undefined behavior"
- Στην πράξη αυτό σημαίνει ότι το πρόγραμμά μας θα κρασάρει (Segmentation Fault) ή ακόμα χειρότερα θα μας χακάρουν



Διάλεξη 11 - Δείκτες και Αναδρομή

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Η Μνήμη Οργανώνεται σε Bytes (Υπενθύμιση)

Το μέγεθος της μνήμης μετράται σε Bytes:

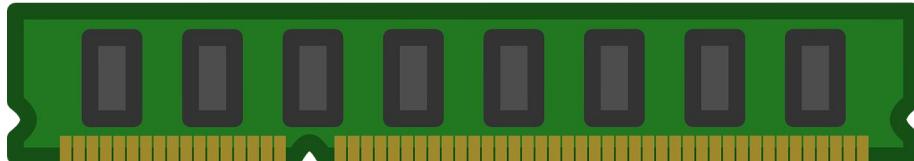
- **1 KB (KiloByte) = 1.000 Bytes**
- **1 MB (MegaByte) = 1.000.000 Bytes**
- **1 GB (GigaByte) = 1.000.000.000 Bytes**

Μνήμη με
N Bytes

Byte 0
Byte 1
Byte 2
...
Byte N-1
Byte N

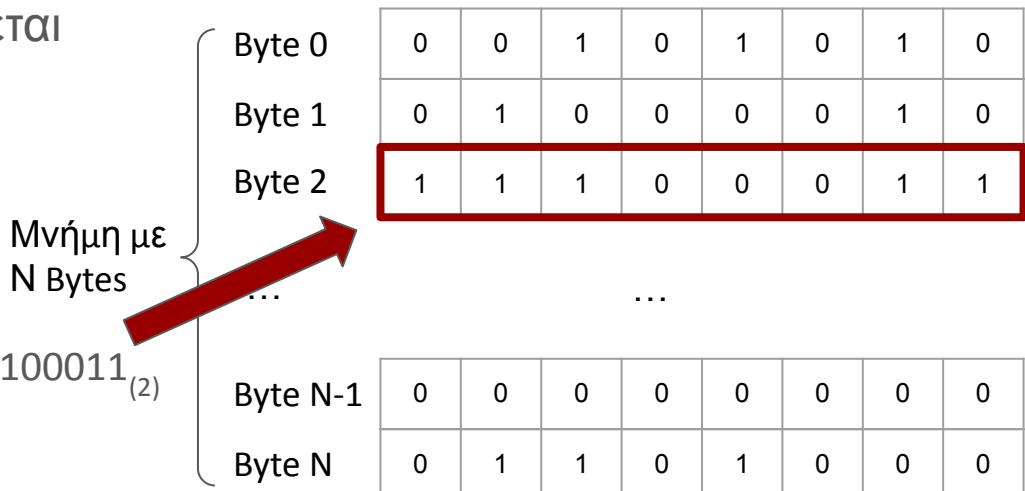
0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
1	1	1	0	0	0	1	1
...							

0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0

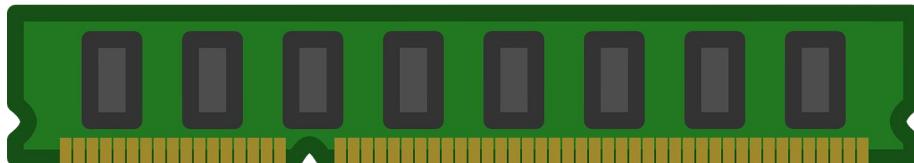


Διεύθυνση ενός Κελιού Μνήμης

Η θέση ενός κελιού στην μνήμη λέγεται **διεύθυνση (address)**.



πχ: στην διεύθυνση 2 υπάρχει το byte $11100011_{(2)}$



Δήλωση Μεταβλητής (Variable Declaration)

Μεταβλητή είναι ένα τμήμα της μνήμης με συγκεκριμένο όνομα.

Η μεταβλητή για να χρησιμοποιηθεί πρέπει να έχει δηλωθεί με κάποιον τύπο.

π.χ. 4 bytes
για την x
ξεκινώντας
από το 0

int x;

Ο τύπος (type) της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για τα περιεχόμενα

Το όνομα (name) της μεταβλητής κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης που θα την αποθηκεύσει

Byte 0	0	0	1	0	1	0	1	0
Byte 1	0	1	0	0	0	0	1	0
Byte 2	1	1	1	0	0	0	1	1
Byte 3	1	1	1	0	0	0	1	1
...	...							
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

Ανάθεση σε Μεταβλητή (Variable Assignment)

Ανάθεση σε μια μεταβλητή μπορεί να γίνει κατά τον ορισμό της:

```
int x = 42;
```

Ή μετά τον ορισμό της:

```
int x;
```

```
x = 42;
```

Ή με δεκαεξαδικό τρόπο:

```
int x = 0x2A;
```

Περιεχόμενο της x
πριν την ανάθεση

Byte 0	0	0	1	0	1	0	1	0
Byte 1	0	1	0	0	0	0	1	0
Byte 2	1	1	1	0	0	0	1	1
Byte 3	1	1	1	0	0	0	1	1
...
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

Ανάθεση σε Μεταβλητή (Variable Assignment)

Ανάθεση σε μια μεταβλητή μπορεί να γίνει κατά τον ορισμό της:

```
int x = 42;
```

Ή μετά τον ορισμό της:

```
int x;
```

```
x = 42;
```

Ή με δεκαεξαδικό τρόπο:

```
int x = 0x2A;
```

Περιεχόμενο της x
μετά την ανάθεση

Byte 0	0	0	0	0	0	0	0	0
Byte 1	0	0	0	0	0	0	0	0
Byte 2	0	0	0	0	0	0	0	0
Byte 3	0	0	1	0	1	0	1	0
...	...							
Byte N-1	0	0	0	0	0	0	0	0
Byte N	0	1	1	0	1	0	0	0

Διεύθυνση (Address) Μιας Μεταβλητής

Μπορούμε να βρούμε την **διεύθυνση** μιας μεταβλητής χρησιμοποιούμε τον μοναδιαίο τελεστή & (ampersand):

```
int x = 42;
```

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

Όταν το τρέξουμε:

```
$ ./test
```

```
100
```

&x

...

Byte 100

Byte 101

Byte 102

Byte 103

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0

...

...

Η διεύθυνση είναι πάντα ένας **ακέραιος** αριθμός (100 στο παράδειγμα). Ο αριθμός των bits για την αναπαράσταση μιας διεύθυνσης καθορίζεται από τον μεταγλωττιστή, για παράδειγμα 32-bit για συστήματα των 32-bit (-m32), 64-bit για συστήματα των 64-bit κοκ.

Διεύθυνση (Address) Μιας Μεταβλητής

Μπορούμε να βρούμε την **διεύθυνση** μιας μεταβλητής χρησιμοποιούμε τον μοναδιαίο τελεστή & (ampersand):

```
int x = 42;
```

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

Όταν το τρέξουμε:

```
$ ./test
```

```
100
```

&x

...

Byte 100

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0

Byte 101

Byte 102

Byte 103

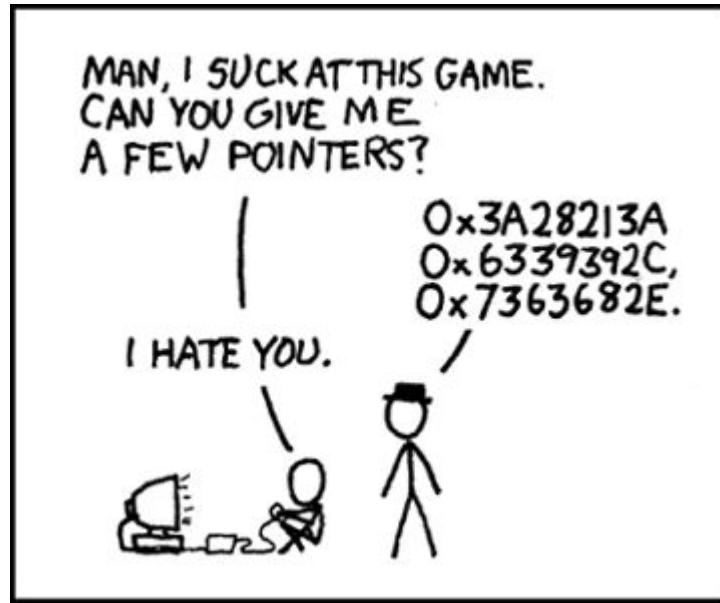
...

...

Σε αυτήν την εκτέλεση του προγράμματος, η μεταβλητή x τοποθετήθηκε στην διεύθυνση 100. Προσοχή: οι διευθύνσεις μπορούν να αλλάξουν από εκτέλεση σε εκτέλεση.

Θα προσθέσουμε έναν νέο τύπο (δείκτη / pointer) για να αποθηκεύουμε **διευθύνσεις** μεταβλητών

Πέρα από τους βασικούς int, char, double



Δήλωση Τύπου Δείκτη (Pointer) στην C

Ο δείκτης είναι μία μεταβλητή που περιέχει την διεύθυνση μνήμης ενός συγκεκριμένου τύπου δεδομένων. Γενική μορφή:

τύπος * όνομα;

Ο τύπος * λέει στον μεταγλωτιστή ότι η διεύθυνση του δείκτη είναι για δεδομένα τύπου τύπος

Το όνομα (name) της μεταβλητής που κρατάει την τιμή του δείκτη - ο μεταγλωτιστής επιλέγει που θα αποθηκευτεί

Δήλωση Τύπου Δείκτη (Pointer) στην C

Ο δείκτης είναι μία μεταβλητή που περιέχει την διεύθυνση μνήμης ενός συγκεκριμένου τύπου δεδομένων. Γενική μορφή:

```
int * pointer;
```

O **int** * τύπος λέει στον μεταγλωττιστή ότι η διεύθυνση που αποθηκεύει ο δείκτης είναι για δεδομένα τύπου **int**

Το όνομα (**name**) της μεταβλητής που κρατάει την τιμή του δείκτη - ο μεταγλωττιστής επιλέγει που θα αποθηκευτεί

Αρχικοποίηση ενός Δείκτη σε int

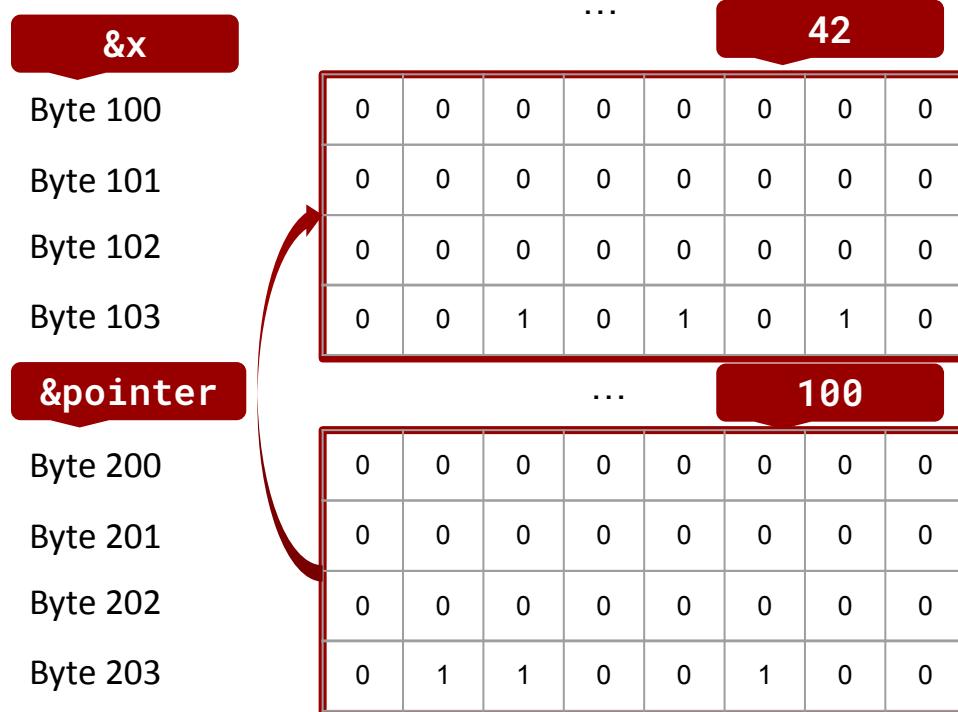
Αρχικοποιούμε έναν δείκτη με την διεύθυνση μιας μεταβλητής ως εξής:

```
int x = 42;  
int *pointer = &x;  
printf("%d, %d\n", pointer, &pointer);
```

Όταν το τρέξουμε:

```
$ ./test  
100, 200
```

Λέμε ότι ο pointer δείχνει (points to) στην μεταβλητή x.



Αρχικοποίηση ενός Δείκτη σε char

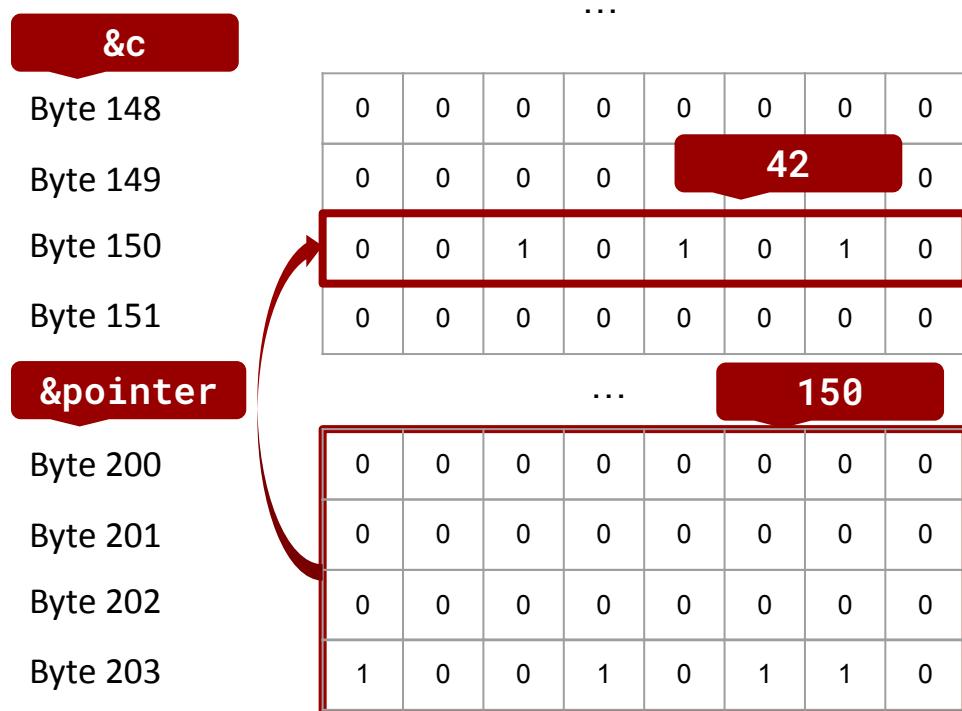
Αρχικοποιούμε έναν δείκτη με την διεύθυνση ενός χαρακτήρα ως εξής:

```
char c = 42;  
char *pointer = &c;  
printf("%d, %d\n", pointer, &pointer);
```

Όταν το τρέξουμε:

```
$ ./test  
150, 200
```

Λέμε ότι ο pointer δείχνει (points to) στην μεταβλητή x.



Ο τελεστής `sizeof`

Υπολογίζει τον αριθμό των bytes που δεσμεύει στην μνήμη του υπολογιστή ο τύπος δεδομένων ή η μεταβλητή που δηλώνεται στις παρενθέσεις του

```
printf("int size: %d\n", sizeof(int));
```

Ή πιο "σωστά":

```
printf("int size: %zu\n", sizeof(int));
```

```
./sizeof  
int size: 4
```

Τι θα τυπώσει το παρακάτω πρόγραμμα:

```
#include <stdio.h>

int main() {
    int * ipointer;
    char * cpointer;
    double * dpointer;
    printf("%d %d %d\n", sizeof(ipointer), sizeof(cpointer), sizeof(dpointer));
    return 0;
}
```

Η ειδική τιμή NULL

Όταν θέλουμε να δηλώσουμε ότι ένας δείκτης **δεν δείχνει σε κάποια μεταβλητή**, του αναθέτουμε την τιμή **NULL** (διεύθυνση 0).

```
int * ipointer = NULL;  
...  
if (ipointer == NULL) {  
    printf("pointer does not point anywhere\n");  
}
```

Δεν υπάρχει περίπτωση όμως στην διεύθυνση 0 να υπάρχει μεταβλητή; Θεωρητικά ναι, πρακτικά όχι.

The billion dollar mistake

Ο δείκτης δείχνει σε μια μεταβλητή - μπορώ να προσπελάσω την μεταβλητή έχοντας μόνο την διεύθυνσή της;

Χρήση Δεικτών (Dereference Pointers)

Για να χρησιμοποιήσουμε το περιεχόμενο της μεταβλητής στην οποία δείχνει ένας δείκτης χρησιμοποιούμε τον μοναδιαίο τελεστή * :

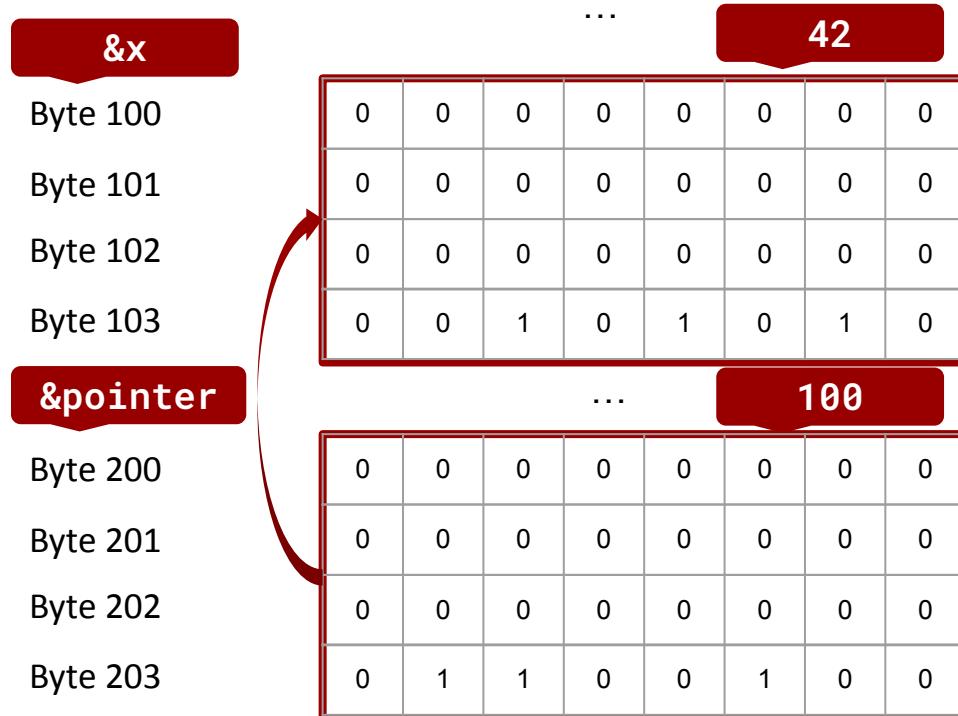
```
int x = 42;  
int *pointer = &x;  
printf("%d\n", *pointer);
```

Όταν το τρέξουμε:

```
$ ./test
```

42

Η χρήση του `*pointer` είναι ισοδύναμη με την χρήση της μεταβλητής `x`.



Χρήση Μη-Έγκυρων Δεικτών (Invalid Pointers)

Για να χρησιμοποιήσουμε το περιεχόμενο της διεύθυνσης στην οποία δείχνει ένας δείκτης, η διεύθυνση πρέπει πρώτα να υπάρχει.

```
int *pointer;  
printf("%d\n", *pointer);
```

Το παραπάνω πρόγραμμα κατά πάσα πιθανότητα θα οδηγήσει σε σφάλμα **segmentation fault**, καθώς το περιεχόμενο της μεταβλητής pointer δεν έχει αρχικοποιηθεί και επομένως δεν θα έχει μια έγκυρη διεύθυνση μνήμης.

Οι τελεστές * και & είναι συμπληρωματικοί

Ο τελεστής * επιστρέφει την μεταβλητή σε μια διεύθυνσης μνήμης, ενώ ο τελεστής & επιστρέφει την διεύθυνση μνήμης μιας μεταβλητής. Επομένως λέμε ότι αυτοί οι δύο τελεστές είναι συμπληρωματικοί (ή αντίστροφοι, ή αλλιώς ότι αλληλοανανεωύνται όταν εφαρμόζονται σε **έγκυρους δείκτες**).

```
int x = 42;  
  
int *pointer = &x;  
  
printf("%p %p %p\n", pointer, &pointer, *&pointer);
```

Τρέχοντας το παραπάνω:

```
$ ./pointer_size  
0x7ffcf94870cc 0x7ffcf94870cc 0x7ffcf94870cc
```

Τι τυπώνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int a = 100, b = 200, c;
    int *ptr_a = &a, *ptr_b = &b, *ptr_c = &c;
    *ptr_c = a;
    *ptr_a = b;
    *ptr_b = *ptr_c;
    printf("%d %d %d", a, b, c);
    return 0;
}
```

Πράξεις με Δείκτες (Διευθύνσεις)

Μπορούμε να εφαρμόσουμε τελεστές σε δείκτες στις ακόλουθες περιπτώσεις:

- Πρόσθεση ή αφαίρεση ακεραίου σε/από δείκτη
- Αφαίρεση δύο δεικτών
- Σύγκριση δύο δεικτών ή με το 0 (NULL)

Πρόσθεση Ακεραίου σε δείκτη

Η πρόσθεση ενός ακεραίου αυξάνει την διεύθυνση του δείκτη κατά το μέγεθος του τύπου στον οποίο δείχνει πολλαπλασιασμένο με τον ακέραιο

τύπος * pointer; pointer += N => pointer = (int)pointer + N * sizeof(τύπος)

Παραδείγματα:

int *ipointer; ipointer += 2; // => + 2 * sizeof(int)

char *cpointer; cpointer += 2; // => + 2 * sizeof(char)

double *dpointer; dpointer += 2; // => + 2 * sizeof(double)

Αύξηση Δείκτη σε int

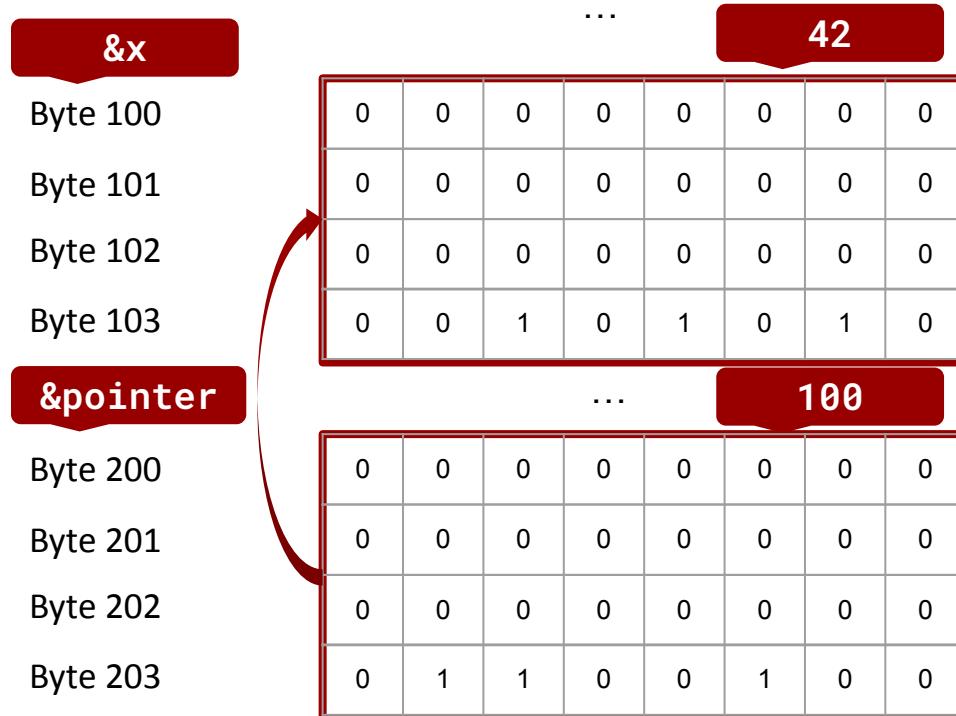
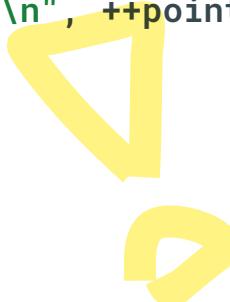
Για να χρησιμοποιήσουμε το περιεχόμενο της μεταβλητής στην οποία δείχνει ένας δείκτης χρησιμοποιούμε τον μοναδιαίο τελεστή * :

```
int x = 42;  
int *pointer = &x;  
printf("%d\n", ++pointer);
```

Όταν το τρέξουμε:

\$./test

104



Μείωση δείκτη σε char

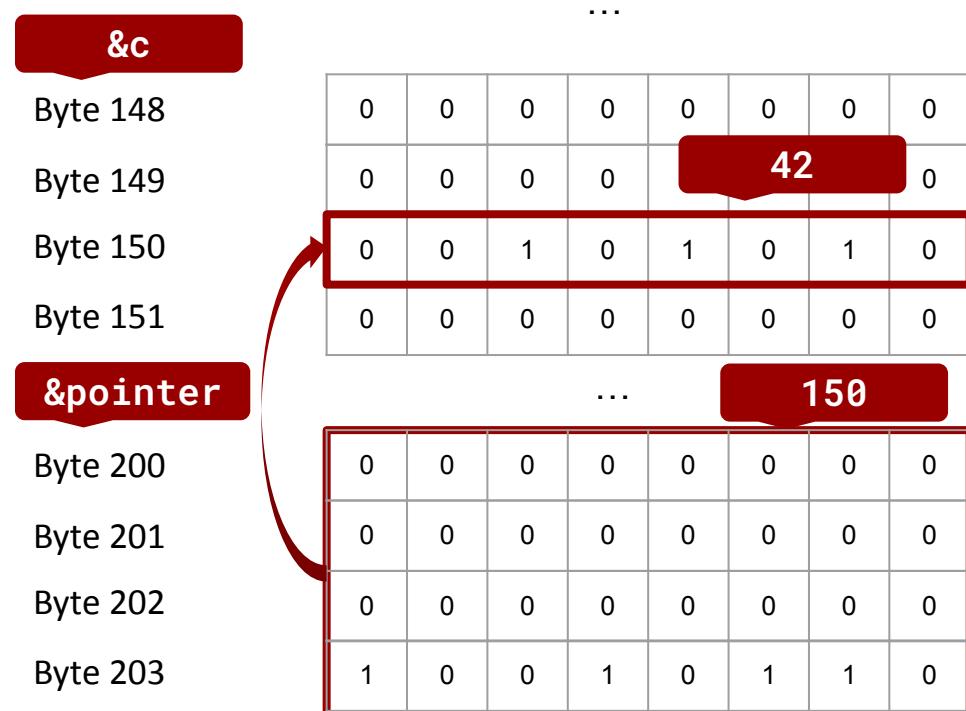
Αρχικοποιούμε έναν δείκτη με την διεύθυνση ενός χαρακτήρα ως εξής:

```
char c = 42;  
char *pointer = &c;  
printf("%d\n", --pointer);
```

Όταν το τρέξουμε:

```
$ ./test
```

```
149
```



Αναφορά σε στοιχεία πίνακα - Τι θα τυπώσει;

```
#include <stdio.h>

int main() {
    int *ptr, arr[ ] = {10, 20, 30};
    ptr = &arr[0];
    printf("mem[%p], %d\n", ptr, *ptr);
    ptr += 2;
    printf("mem[%p], %d\n", ptr, *ptr);
    return 0;
}
```

Αναφορά σε στοιχεία πίνακα - Τι θα τυπώσει;

```
#include <stdio.h>

int main() {
    int *ptr, arr[] = {10, 20, 30};
    ptr = &arr[0];
    printf("mem[%p], %d\n", ptr, *ptr);
    ptr += 2;
    printf("mem[%p], %d\n", ptr, *ptr);
    return 0;
}
```

```
$ ./test
mem[0ffa35e60], 10
mem[0ffa35e68], 30
```

Συντομογραφία για χρήση στοιχείου

Η έκφραση $*(\text{ptr} + n)$ - αύξησε τον δείκτη κατά n στοιχεία και επίστρεψε την μεταβλητή που αντιστοιχεί είναι τόσο συχνή που έχουμε μια συντομογραφία:

$$*(\text{ptr} + n) \Leftrightarrow \text{ptr}[n]$$

$$*(\text{ptr} + 3) \Leftrightarrow \text{ptr}[3]$$

Μας θυμίζει κάτι;



Συντομογραφία για χρήση στοιχείου

Η έκφραση $*(\text{ptr} + n)$ - αύξησε τον δείκτη κατά n στοιχεία και επίστρεψε την μεταβλητή που αντιστοιχεί είναι τόσο συχνή που έχουμε μια συντομογραφία:

$$*(\text{ptr} + n) \Leftrightarrow \text{ptr}[n]$$

$$*(\text{ptr} + 3) \Leftrightarrow \text{ptr}[3]$$

Μας θυμίζει κάτι;

Είναι όμοιο με την έκφραση αναφοράς σε ένα στοιχείο πίνακα!

```
int a[100]; // το a είναι ένας δείκτης κολλημένος στο &a[0]
```

Διαφορές Πινάκων και Δεικτών

Παρόλο που η προσπέλαση στοιχείων είναι η ίδια, και ο πίνακας είναι ουσιαστικά ένας δείκτης στο πρώτο στοιχείο, υπάρχουν διαφορές. Π.χ.:

```
int a[100]; int *ptr;
```

- 1 • Δεν μπορούμε να αλλάξουμε την διεύθυνση ενός πίνακα (`a = ptr`)
- 2 • Η δήλωση ενός πίνακα δημιουργεί θέσεις μνήμης για τα στοιχεία του (100 int), ενώ η δήλωση ενός δείκτη δημιουργεί θέση για μια διεύθυνση
- 3 • Ο τελεστής `sizeof` γυρνάει το μέγεθος του πίνακα (`sizeof(a) == 100 * sizeof(int)`) και όχι το εύρος ενός ακεραίου διεύθυνσης (`sizeof(ptr)`)
- 4 • Ο τελεστής `&` επιστρέφει την διεύθυνση του πρώτου στοιχείου του πίνακα (`&a == &a[0]`) και όχι την διεύθυνση ενός δείκτη (`&ptr`)

Αναδρομή

Η Συνάρτηση Παραγοντικό (Factorial)

Στα μαθηματικά το παραγοντικό ενός φυσικού αριθμού n , συμβολίζεται με $n!$ και είναι το γινόμενο όλων των θετικών ακεραίων μικρότερων ή ίσων του n .

Ο ορισμός του παραγοντικού στα μαθηματικά είναι **αναδρομικός (recursive)**:

$$n! = \begin{cases} 1 & , n \leq 1 \\ n * (n - 1)! & , n > 1 \end{cases}$$

Πως θα το γράφαμε σε C;

```

#include <stdio.h>
#include <stdlib.h>
// Compute the factorial of a number using the recursive
// formula.

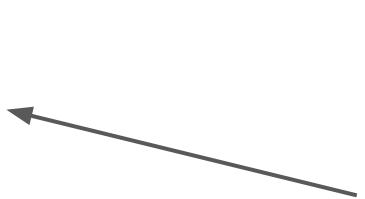
int factorial(int number) {
    if (number <= 1) return 1;
    else return number * factorial(number - 1);
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Program needs to be called as `./prog number`\n");
        return 1;
    }

    int number = atoi(argv[1]);
    printf("%d! = %d\n", number, factorial(number));
    return 0;
}

```

$$n! = \begin{cases} 1 & , n \leq 1 \\ n * (n - 1)! & , n > 1 \end{cases}$$



Η αναδρομική υλοποίηση είναι ιδιαίτερα κοντά στον ορισμό του παραγοντικού - οδηγώντας σε πιο "εύκολο" έλεγχο ορθότητας

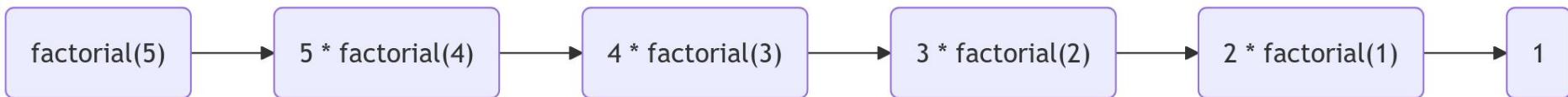
Παρατηρήσεις

- Για κάποιους αριθμούς το αποτέλεσμα είναι αρνητικό. Τι συμβαίνει;

```
$ ./fact 20
```

```
20! = -2102132736
```

- Πόσες αναδρομικές (στον εαυτό της) κλήσεις κάνει η κλήση factorial(5);



- Πόσες αναδρομικές (στον εαυτό της) κλήσεις κάνει η κλήση factorial(N);
- Τι θα συμβεί αν δώσουμε έναν αρνητικό αριθμό στην συνάρτησή μας;

Αναδρομή (Recursion)

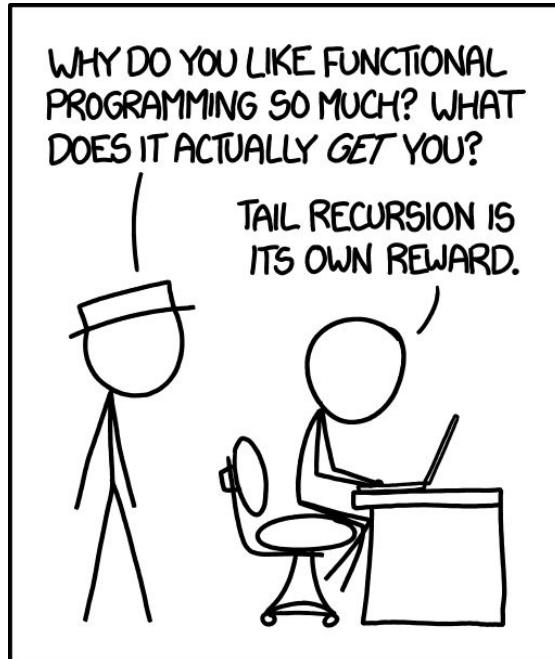
Αναδρομή είναι η μέθοδος κατά την οποία μια συνάρτηση **καλεί τον εαυτό της**, με στόχο να επιλύσει ένα υποπρόβλημα του αρχικού προβλήματος, έως ότου φτάσει σε μια βάση τερματισμού (base case) όπου η αναδρομή σταματά.

Έχει δύο βασικά στοιχεία:

1. **Base case** (Βασική περίπτωση τερματισμού): Μια συνθήκη καθορίζει πότε θα σταματήσει η αναδρομή
2. **Recursive case** (Αναδρομική περίπτωση): Το τμήμα του κώδικα όπου η συνάρτηση καλεί τον εαυτό της

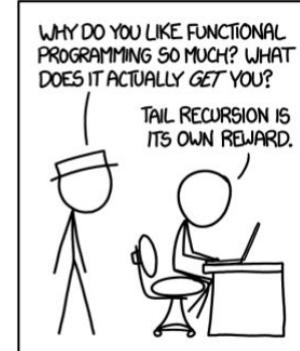
Η Αναδρομή Πρέπει να Τερματίζει

Δεν ξεχνάμε να γράψουμε σωστά Base Cases



Η Αναδρομή Πρέπει να Τερματίζει

Δεν ξεχνάμε να γράψουμε σωστά Base Cases



Η Αναδρομή Πρέπει να Τερματίζει

Δεν ξεχνάμε να γράψουμε σωστά Base Cases



Η Αναδρομή Πρέπει να Τερματίζει

Δεν ξεχνάμε να γράψουμε σωστά Base Cases



Για την επόμενη φορά

- Σε αυτήν και την επόμενη διάλεξη θα καλύψουμε έννοιες από τις σελίδες 73-103 από τις σημειώσεις του κ. Σταματόπουλου.
- Διατρέξτε όποιο tutorial μπορείτε να βρείτε σε pointers [\[1\]](#), [\[2\]](#), [\[3\]](#), [\[4\]](#)

Ευχαριστώ και καλή μέρα εύχομαι!
Keep Coding ;)

Διάλεξη 12 - Δείκτες και Πίνακες - 2D+

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

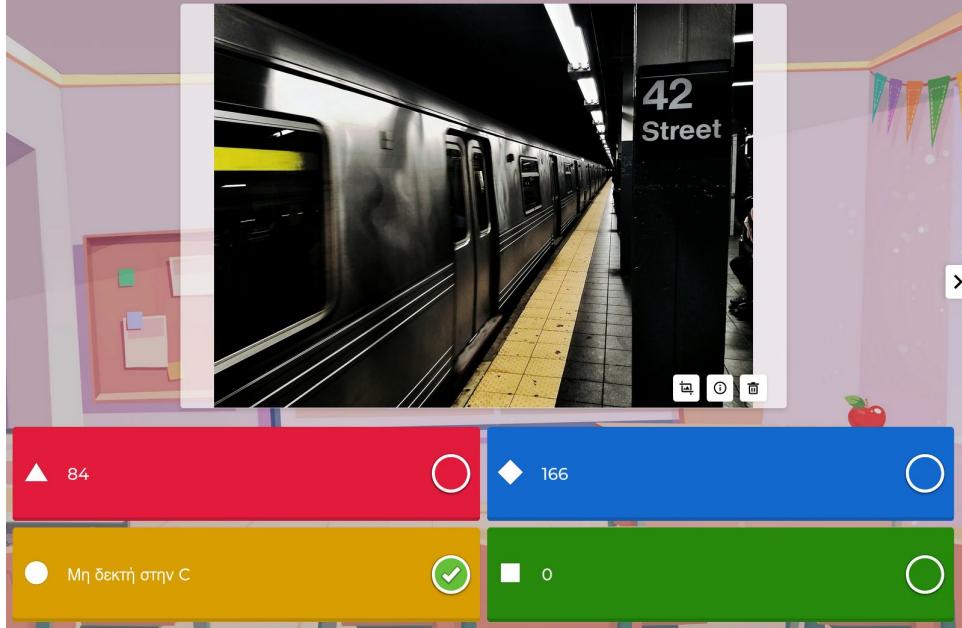
Θανάσης Αυγερινός

Ανακοινώσεις / Διευκρινήσεις

- Η Εργασία #1 μόλις βγήκε! Προθεσμία: Τρίτη, 17 Δεκέμβρη 23:59
- Αναπλήρωση: Πιθανή ημερομηνία 28 Νοέμβρη, 7μμ-9μμ
- Πράξεις με δείκτες

Μπορώ να προσθέσω pointers;

Έστω ότι `int * p, * q` είναι δύο δείκτες και ο καθένας έχει τιμή **42**.
Τότε η παράσταση `p + q` είναι:



▲ 84



◆ 166



● Μη δεκτή στην C



■ 0



```
int main() {  
    int * p = 42, * q = 42;  
  
    return (int)(p + q);  
}  
  
$ gcc -o ptr ptr.c  
...  
ptr.c:3:18: error: invalid  
operands to binary + (have 'int *'  
and 'int *')  
3 |     return (int)(p + q);
```

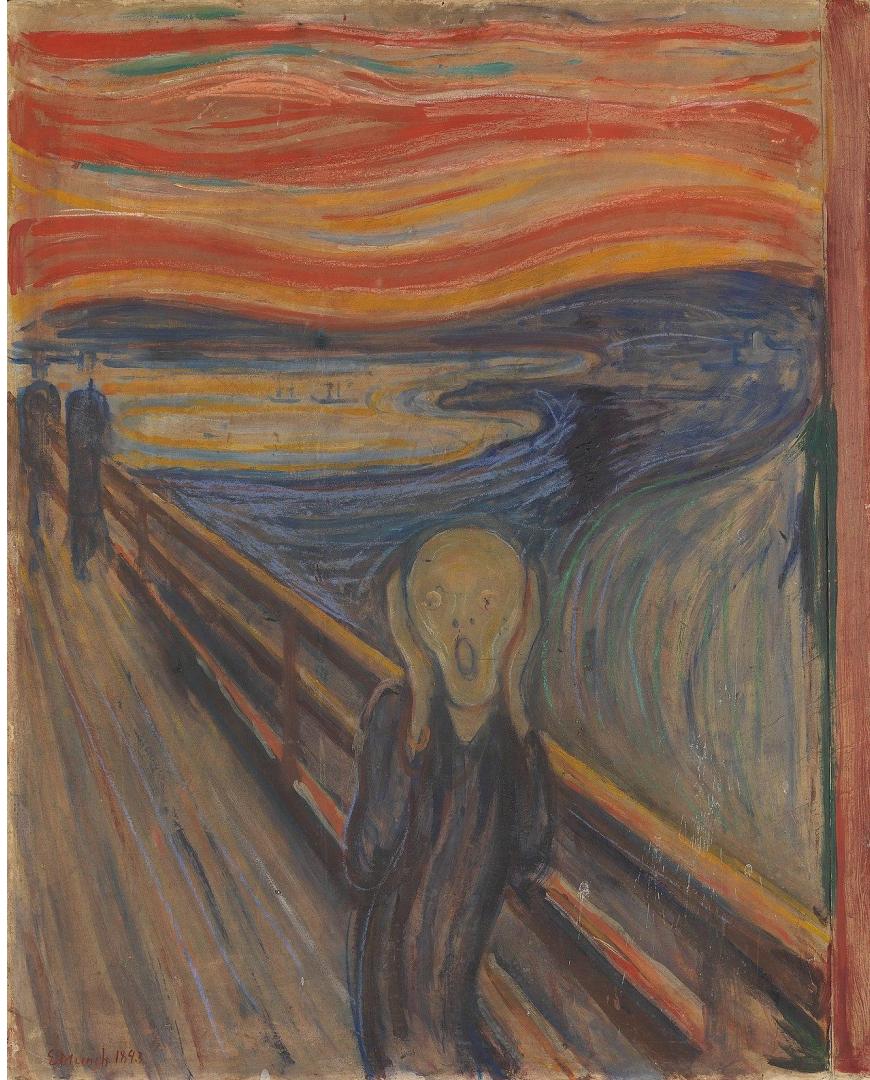
Πράξεις με Δείκτες (Διευθύνσεις)

Μπορούμε να εφαρμόσουμε τελεστές σε δείκτες στις ακόλουθες περιπτώσεις:

- Πρόσθεση ή αφαίρεση ακεραίου σε/από δείκτη
- Αφαίρεση δύο δεικτών
- Σύγκριση δύο δεικτών ή με το 0 (NULL)

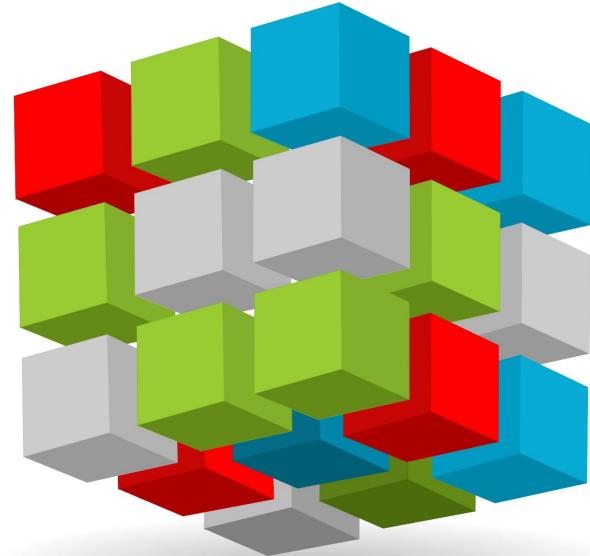
Την Προηγούμενη Φορά

- Δείκτες (Pointers)
 - Διευθύνσεις μνήμης
 - Χρήση δεικτών
 - Πράξεις με δείκτες
 - Παραδείγματα
- Παραδείγματα
- Αναδρομή



Σήμερα

- Δείκτες και Πίνακες reloaded
 - Δισδιάστατοι πίνακες
 - Παραδείγματα
 - Δείκτες σε δείκτες σε δείκτες ...
 - Δυναμική διαχείριση μνήμης



Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειρίζομαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int array[100];
```

Ο **τύπος (type)** της μεταβλητής λέει στον μεταγλωττιστή πόση μνήμη να δεσμεύσει για κάθε στοιχείο του πίνακα

Το **όνομα (name)** του πίνακα κάνει τον μεταγλωττιστή να διαλέξει την διεύθυνση της μνήμης θα τον αποθηκεύσει

Το **μέγεθος (size)** του πίνακα λέει στον μεταγλωττιστή πόσες θέσεις αυτού του τύπου να κρατήσει - στατικό: αφού δηλωθεί δεν αλλάζει κατά την εκτέλεση

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **Θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Bytes 0-3

Bytes 4-7

Bytes 8-11

Bytes 12-15

Bytes 400-403

Μνήμη

b0	b1	b2	b3
b4	b5	b6	b7
b8	b9	b10	b11
b12	b13	b14	b15
...			
b400	b401	b402	b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

bears[0]	Μνήμη			
Bytes 0-3	b0 b1 b2 b3			
Bytes 4-7	b4	b5	b6	b7
Bytes 8-11	b8	b9	b10	b11
Bytes 12-15	b12	b13	b14	b15
...				
Bytes 400-403	b400	b401	b402	b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Μνήμη			
bears[1]	b0	b1	b2
Bytes 4-7	b4	b5	b6
Bytes 8-11	b8	b9	b10
Bytes 12-15	b12	b13	b14
	...		
Bytes 400-403	b400	b401	b402
			b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειριζόμαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση ενός πίνακα έχει την μορφή:

```
int bears[100];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Μνήμη	b0	b1	b2	b3
Bytes 0-3	b4	b5	b6	b7
Bytes 4-7	b8	b9	b10	b11
Bytes 8-11	b12	b13	b14	b15
bears[99]	...			
Bytes 400-403	b400	b401	b402	b403

Δήλωση Πίνακα (Array) στην Γλώσσα C

Ένας **πίνακας** μας επιτρέπει να χειρίζομαστε ένα σύνολο από δεδομένα ίδιου τύπου με ενιαίο και γενικό τρόπο. Στην C η δήλωση πίνακα γίνεται σε μορφή:

```
int bears[100];
```

Ο πίνακας bears
καταλαμβάνει $4 * 100 = 400$ bytes
μνήμης στις διευθύνσεις 4-403

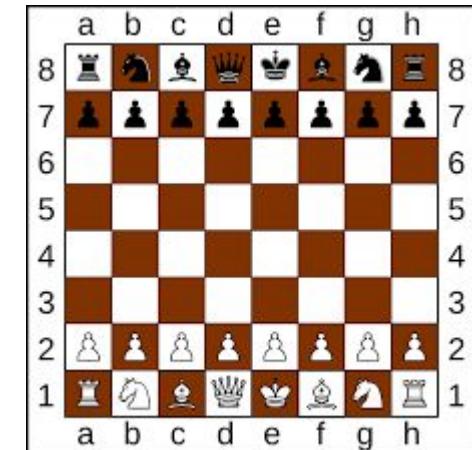
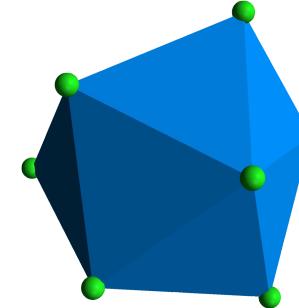
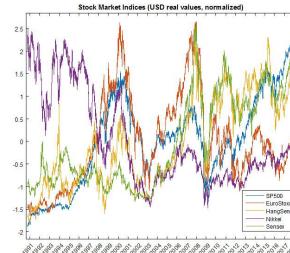
Μπορούμε να αναφερθούμε στο κάθε στοιχείο του πίνακα χρησιμοποιώντας την **θέση (index)** του στοιχείου στον πίνακα:
bears[0], bears[1], ..., bears[99]

Bytes 400-403

Μνήμη			
b0	b1	b2	b3
b4	b5	b6	b7
b8	b9	b10	b11
b12	b13	b14	b15
...			
b400	b401	b402	b403

Κάποια Δεδομένα Αναπαρίστανται Ευκολότερα σε πάνω από μία Διάσταση (Dimension)

- Επεξεργασία εικόνας και γραφικά
- Πολυδιάστατα δεδομένα σε επιστημονικά πεδία (φυσική, χημεία, κτλ)
- Ανάλυση οικονομικών δεδομένων
- Και πολλά άλλα



Δήλωση Δισδιάστατου Πίνακα (Array) στην Γλώσσα C

Ένας δισδιάστατος πίνακας είναι ένας πίνακας από (υπο)πίνακες. Στην C η δήλωση ενός δισδιάστατου πίνακα έχει την μορφή:

τύπος όνομα [γραμμές] [στήλες] ;

Ο τύπος (type) περιγράφει τον τύπο κάθε στοιχείου του πίνακα

Το όνομα (name) του πίνακα που μας επιτρέπει να αναφερόμαστε σε αυτόν

Ο αριθμός των γραμμών (rows) του πίνακα περιγράφει πόσους υποπίνακες (γραμμές) έχει ο πίνακας (1η διάσταση)

Ο αριθμός των στηλών (columns) του πίνακα περιγράφει πόσα στοιχεία έχει ο κάθε υποπίνακας (2η διάσταση)

Σύνολο ο πίνακας έχει γραμμές x στήλες στοιχεία

Δήλωση Δισδιάστατου Πίνακα (Array) στην Γλώσσα C

Ένας **δισδιάστατος πίνακας** είναι ένας πίνακας από (υπο)πίνακες. Ένα παράδειγμα πίνακα 2 γραμμών και 4 στηλών (8 στοιχεία σύνολο) είναι:

```
int array[2][4];
```

Μπορούμε να αναφερθούμε στο κάθε στοιχείο με την έκφραση $a[i][j]$, όπου i είναι η θέση του στοιχείου στην γραμμή και j η θέση του στην στήλη.

	Στήλη 0	Στήλη 1	Στήλη 2	Στήλη 3
Γραμμή 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Γραμμή 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]

Αρχικοποίηση Δισδιάστατου Πίνακα

Η αρχικοποίηση δισδιάστατων πινάκων είναι όπως στους μονοδιάστατους - αντί για στοιχεία τύπου, χρησιμοποιούμε πίνακες:

```
int array[2][4] = {  
    {1, 4, 7, 10},  
    {3, 6, 9, 12},  
};
```

Ποιο είναι το στοιχείο
a[1][1];

Στήλη 0

Στήλη 1

Στήλη 2

Στήλη 3

Γραμμή 0

Γραμμή 1

1	4	7	10
3	6	9	12

Αναπαράσταση Δισδιάστατου Πίνακα στην Μνήμη

Έστω ότι `sizeof(int) == 4`, τότε μια αναπαράσταση του πίνακα θα είναι:

```
int array[2][4] = {  
    {1, 4, 7, 10},  
    {3, 6, 9, 12},  
};
```

Bytes 100-103	1
Bytes 104-107	4
Bytes 108-111	7
Bytes 112-115	10
Bytes 116-119	3
Bytes 120-123	6
Bytes 124-127	9
Bytes 128-131	12

Αναπαράσταση Δισδιάστατου Πίνακα στην Μνήμη

Έστω ότι `sizeof(int) == 4`, τότε μια αναπαράσταση του πίνακα θα είναι:

```
int array[2][4] = {  
    {1, 4, 7, 10},  
    {3, 6, 9, 12},  
};
```

```
printf("%d\n", sizeof(array[0]));
```

```
$ ./test
```

array[0]

Bytes 100-103

1

Bytes 104-107

4

Bytes 108-111

7

Bytes 112-115

10

Bytes 116-119

3

Bytes 120-123

6

Bytes 124-127

9

Bytes 128-131

12

Αναπαράσταση Δισδιάστατου Πίνακα στην Μνήμη

Έστω ότι `sizeof(int) == 4`, τότε μια αναπαράσταση του πίνακα θα είναι:

```
int array[2][4] = {  
    {1, 4, 7, 10},  
    {3, 6, 9, 12},  
};  
  
printf("%d\n", sizeof(array[1]));  
  
$ ./test
```

Bytes 100-103	1
Bytes 104-107	4
Bytes 108-111	7
array[1]	15
Bytes 116-119	3
Bytes 120-123	6
Bytes 124-127	9
Bytes 128-131	12

Αναπαράσταση Δισδιάστατου Πίνακα στην Μνήμη

Έστω ότι `sizeof(int) == 4`, τότε μια αναπαράσταση του πίνακα θα είναι:

```
int array[2][4] = {  
    {1, 4, 7, 10},  
    {3, 6, 9, 12},  
};
```

```
printf("%d\n", sizeof(array));
```

```
$ ./test
```

array

Bytes 100-103

1

Bytes 104-107

4

Bytes 108-111

7

Bytes 112-115

10

Bytes 116-119

3

Bytes 120-123

6

Bytes 124-127

9

Bytes 128-131

12

Υπολογισμός διεύθυνσης στοιχείου στην μνήμη

Έστω η ακόλουθη δήλωση πίνακα:

```
int array[X][Y];
```

Η διεύθυνση του $a[i][j]$ ($\&a[i][j]$) θα είναι:

```
StartAddressOfArray +
```

Bytes 100-103	1
Bytes 104-107	4
Bytes 108-111	7
Bytes 112-115	10
Bytes 116-119	3
Bytes 120-123	6
Bytes 124-127	9
Bytes 128-131	12

Υπολογισμός διεύθυνσης στοιχείου στην μνήμη

Έστω η ακόλουθη δήλωση πίνακα:

```
int array[X][Y];
```

Η διεύθυνση του $a[i][j]$ ($\&a[i][j]$) θα είναι:

```
StartAddressOfArray +
    i * Y * sizeof(int) +
    j * sizeof(int)
```

Bytes 100-103	1
Bytes 104-107	4
Bytes 108-111	7
Bytes 112-115	10
Bytes 116-119	3
Bytes 120-123	6
Bytes 124-127	9
Bytes 128-131	12

Υπολογισμός διεύθυνσης στοιχείου στην μνήμη

Έστω η ακόλουθη δήλωση πίνακα:

```
int array[X][Y];
```

Η διεύθυνση του $a[i][j]$ ($\&a[i][j]$) θα είναι:

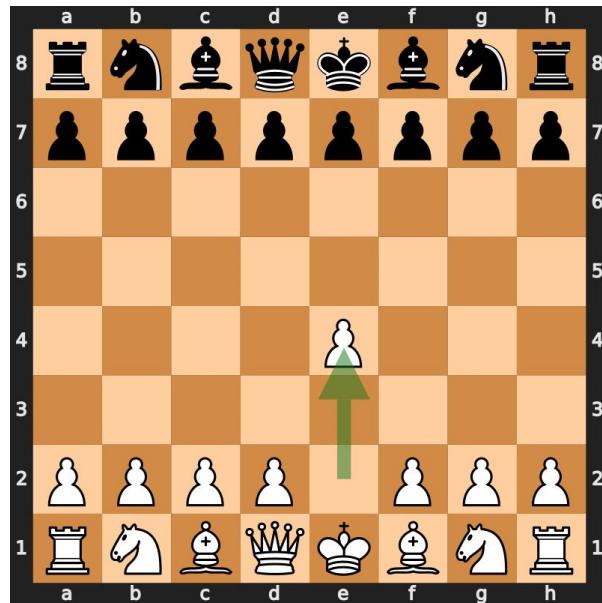
```
IndexOfElementIJ = i * Y + j
```

Παρατηρήστε ότι απλά χρειαζόμαστε τον αριθμό Y των στηλών για να υπολογίσουμε την θέση του $a[i][j]$

Bytes 100-103	1
Bytes 104-107	4
Bytes 108-111	7
Bytes 112-115	10
Bytes 116-119	3
Bytes 120-123	6
Bytes 124-127	9
Bytes 128-131	12

Ανάθεση σε Στοιχεία Δισδιάστατου Πίνακα

```
char chessBoard[8][8] = {  
  
    {'R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'},  
  
    {'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'},  
  
    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},  
  
    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},  
  
    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},  
  
    {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},  
  
    {'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'},  
  
    {'r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'}  
};  
  
chessBoard[1][4] = ' ';  
  
chessBoard[3][4] = 'p';
```

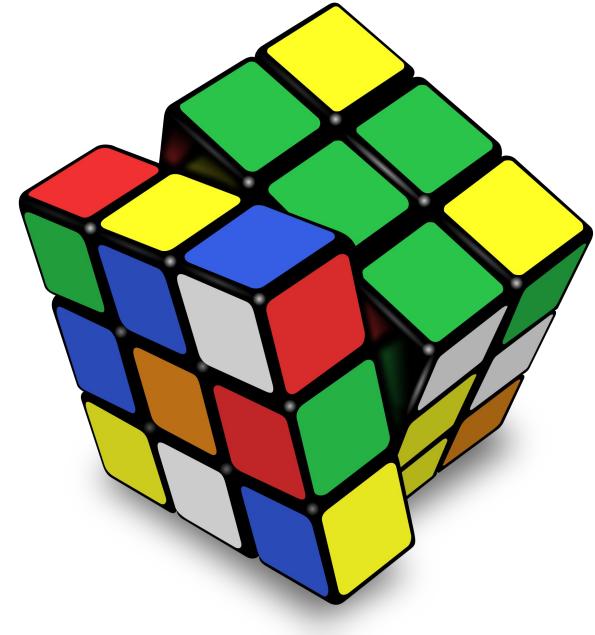


Κάθε array[i][j] στοιχείο μπορεί να χρησιμοποιηθεί ως μεταβλητή

Τρισδιάστατοι, Τετραδιάστατοι, κοκ

Πίνακες υψηλότερων διαστάσεων λειτουργούν με τον ίδιο τρόπο, προσθέτοντας τον επιθυμητό αριθμό διαστάσεων.

```
int rubiksCube[6][3][3] = {  
    {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}}, // Face 1 (e.g., Red)  
    {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}}, // Face 2 (e.g., Green)  
    {{2, 2, 2}, {2, 2, 2}, {2, 2, 2}}, // Face 3 (e.g., Blue)  
    {{3, 3, 3}, {3, 3, 3}, {3, 3, 3}}, // Face 4 (e.g., Yellow)  
    {{4, 4, 4}, {4, 4, 4}, {4, 4, 4}}, // Face 5 (e.g., Orange)  
    {{5, 5, 5}, {5, 5, 5}, {5, 5, 5}} // Face 6 (e.g., White)  
};
```



Είναι απαραίτητοι οι πολυδιάστατοι
πίνακες;

Χρήση Δεικτών (Dereference Pointers) - Υπενθύμιση

Για να χρησιμοποιήσουμε το περιεχόμενο της μεταβλητής στην οποία δείχνει ένας δείκτης χρησιμοποιούμε τον μοναδιαίο τελεστή * :

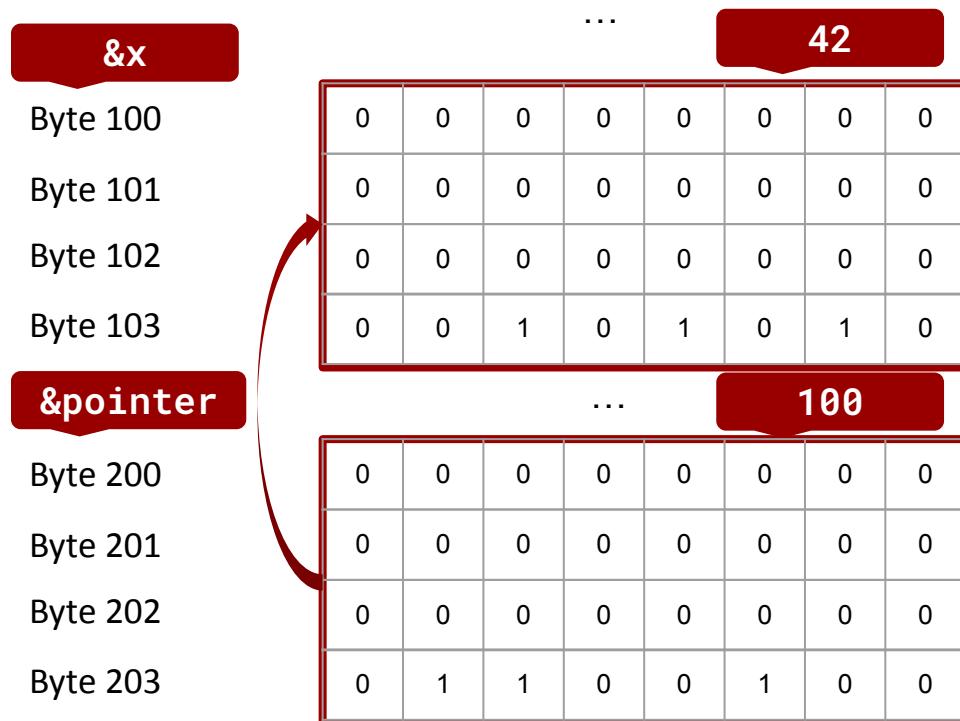
```
int x = 42;  
int *pointer = &x;  
printf("%d\n", *pointer);
```

Όταν το τρέξουμε:

```
$ ./test
```

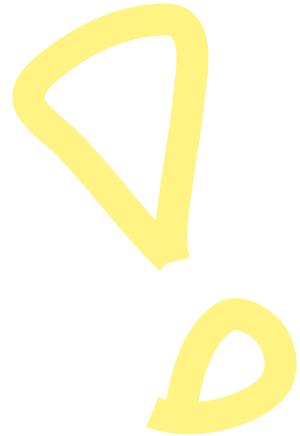
```
42
```

Η χρήση του `*pointer` είναι ισοδύναμη με την χρήση της μεταβλητής `x`.



Ποια είναι τα περιεχόμενα του x μετά την εκτέλεση;

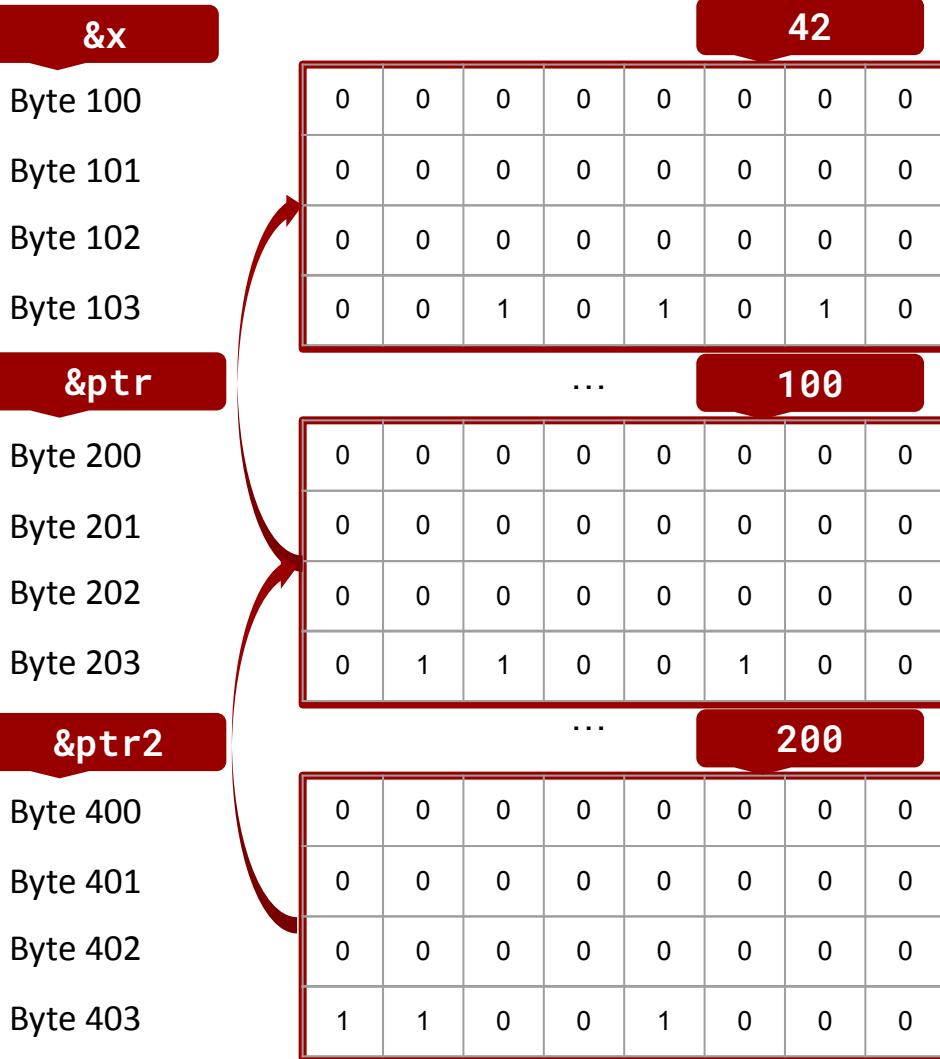
```
int * p;  
int x[ ] = {5, 7, 2, 3, 6, 0, 1, 4};  
p = x;  
while (*p = *(p+2))  
    p++;
```



Δείκτης σε Δείκτη

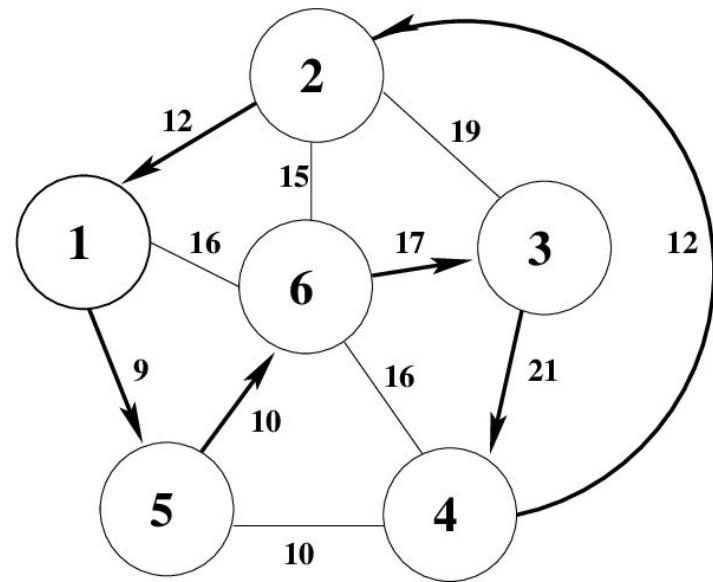
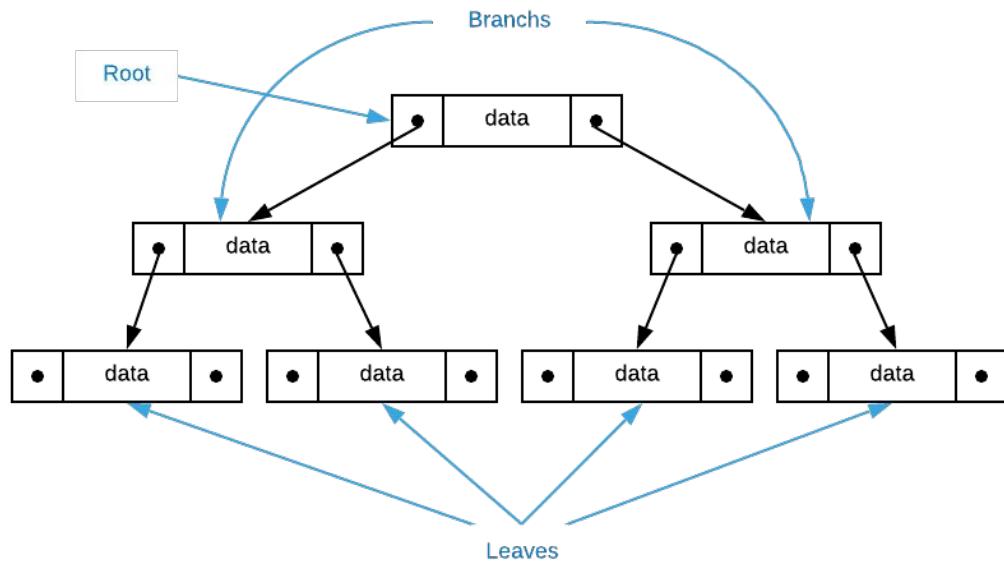
```
int x = 42;  
  
int * ptr = &x;  
  
int ** ptr2 = &ptr;  
  
printf("%p %d", *ptr2, **ptr2);
```

Τι θα τυπώσει το παραπάνω;



Δείκτες σε Δείκτες σε Δείκτες ...

Γιατί έχει σημασία;



Πίνακες από Δείκτες

Οι δείκτες είναι μεταβλητές και μπορούν να μπουν σε πίνακες. Για παράδειγμα τι θα τυπώσει το ακόλουθο:

```
#include <stdio.h>

int main() {
    int *ptr[3], a = 100, b = 200, c = 300;
    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;
    printf("%d %d %d\n", *ptr[2], *ptr[1], *ptr[0]);
    return 0;
}
```

Πίνακες από Δείκτες

Οι δείκτες είναι μεταβλητές και μπορούν να μπουν σε πίνακες. Για παράδειγμα τι θα τυπώσει το ακόλουθο:

```
#include <stdio.h>

int main() {
    int *ptr[3], a = 100, b = 200, c = 300;
    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;
    printf("%d %d %d\n", *ptr[2], *ptr[1], *ptr[0]);
    return 0;
}
```

\$./example
300 200 100

Πίνακες από Δείκτες

```
#include <stdio.h>

int main() {
    char *sentence[] = {
        "I'm", "singing", "in", "the", "rain", "!", NULL
    };
    int i;
    for(i = 0 ; sentence[i]; i++) {
        printf("%s\n", sentence[i]);
    }
    return 0;
}
```

Πίνακες από Δείκτες

```
#include <stdio.h>

int main() {
    char *sentence[] = {
        "I'm", "singing", "in", "the", "rain", "!", NULL
    };
    int i;
    for(i = 0 ; sentence[i]; i++) {
        printf("%s\n", sentence[i]);
    }
    return 0;
}
```

```
$ ./sentence
I'm
singing
in
the
rain
!
```

To περίφημο argv

```
#include <stdio.h>

int main(int argc, char * argv[]) {
    int i;
    for(i = 0 ; i < argc ; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

To περίφημο argv

```
#include <stdio.h>

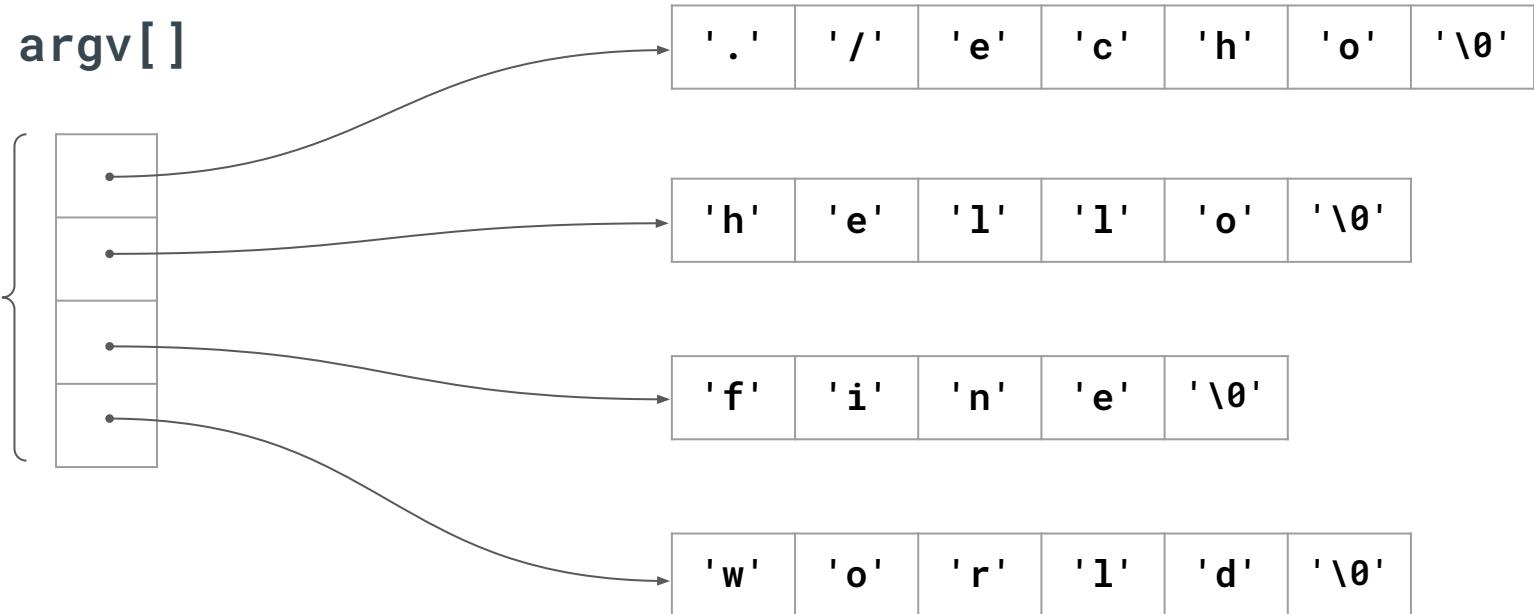
int main(int argc, char * argv[]) {
    int i;
    for(i = 0 ; i < argc ; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

```
$ ./echo hello fine world
./echo
hello
fine
world
```

To περίφημο argv

`char * argv[]`

`argc = 4`



Δυναμικοί Πίνακες με την συνάρτηση malloc()

Με την βοήθεια των δεικτών, μπορούμε να χρησιμοποιήσουμε **δυναμικούς** πίνακες, πίνακες των οποίων το μέγεθος καθορίζεται **δυναμικά**, δηλαδή την στιγμή που τρέχει το πρόγραμμα. Γενική μορφή:

τύπος * όνομα = malloc(μέγεθος * sizeof(τύπος));

Δήλωση δείκτη σε δεδομένα **τύπος** (**type**) - περιγράφει τον τύπο κάθε στοιχείου του πίνακα

Η συνάρτηση **malloc** επιστρέφει την διεύθυνση της μνήμης που θα βάλουμε τα στοιχεία του πίνακα

Ο αριθμός των **bytes** του πίνακα περιγράφει πόσος χώρος πρέπει να δεσμευτεί για να χωρέσει **μέγεθος** φορές τον τύπο

Δυναμικοί Πίνακες με την συνάρτηση malloc()

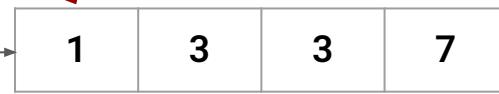
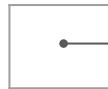
Με την βοήθεια των δεικτών, μπορούμε να χρησιμοποιήσουμε **δυναμικούς** πίνακες, πίνακες των οποίων το μέγεθος καθορίζεται **δυναμικά**, δηλαδή την στιγμή που τρέχει το πρόγραμμα. Παράδειγμα:

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

Η μνήμη που επιστρέφει η συνάρτηση malloc
δεσμεύεται σε ένα μέρος της μνήμης που λέγεται
σωρός (heap) - επόμενο μάθημα

Δημιουργία
πίνακα N
ακεραίων

```
int * array
```



array[0]

array[N - 1]

Δυναμικοί Πίνακες με την συνάρτηση malloc()

```
int * nums = malloc(100 * sizeof(int));  
  
double * coeffs = malloc(100 * sizeof(double));  
  
char * str = malloc(100 * sizeof(char));
```

Πόση μνήμη δεσμεύεται με καθεμιά από τις παραπάνω κλήσεις;

Τι θα τυπώσει το ακόλουθο:

```
printf("%d %d %d\n", sizeof(nums), sizeof(coeffs), sizeof(str));
```

```
$ ./dynamic  
8 8 8
```

Διάλεξη 13 - Μνήμη

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Κατηγορίες Μνήμης

Υπάρχουν 3 κατηγορίες μνήμης:

1. Η στοίβα (stack)
2. Ο σωρός (heap)
3. Η παγκόσμια / στατική μνήμη (global / static memory) - όχι σήμερα

Η στοίβα (Stack)

Η στοίβα (stack) είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.



Η στοίβα (Stack)

Η στοίβα (stack) είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62; char c = 63;
```

Byte 1	
Byte 2	
Byte 3	
Byte 31996	
Byte 31997	
Byte 31998	63
Byte 31999	62
Byte 32000	61

Η στοίβα (Stack)

Η στοίβα (stack) είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62; char c = 63;
```

```
...
```

```
char d = 64;
```

Byte 1

Byte 2

Byte 3

Byte 31996

Byte 31997

Byte 31998

63

Byte 31999

62

Byte 32000

61

Η στοίβα (Stack)

Η στοίβα (stack) είναι μια **συνεχόμενη** περιοχή της μνήμης που προστίθενται και αφαιρούνται στοιχεία με σειρά **Last-In-First-Out (LIFO)**, δηλαδή το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που θα αφαιρεθεί.

```
char a = 61; char b = 62;
```

Byte 1

Byte 2

Byte 3

Byte 31996

Byte 31997

Byte 31998

Byte 31999

62

Byte 32000

61

Τι αποθηκεύεται συνήθως στην στοίβα;

1. Οι τοπικές μεταβλητές που χρησιμοποιεί κάθε κλήση συνάρτησης
2. Τα ορίσματα που περνάμε στην κλήση συνάρτησης
3. Προσωρινά δεδομένα που αποθηκεύει ο μεταγλωττιστής για κάθε κλήση συνάρτησης

Τι αποθηκεύεται συνήθως στην στοίβα;

2: Ορίσματα που περνάμε στην συνάρτηση

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

1: Τοπικές
Μεταβλητές
ορισμένες
μέσα στην
συνάρτηση

3: Προσωρινά δεδομένα (συνήθως
μερικά bytes) που αποθηκεύει ο
μεταγλωττιστής

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

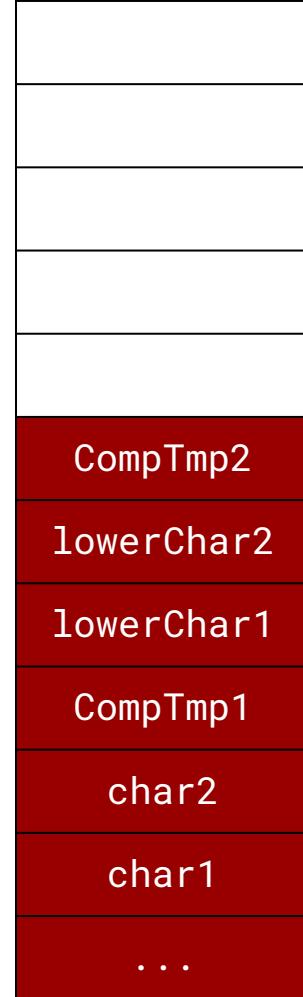
char1

...

Τι αποθηκεύεται συνήθως στην στοίβα;

```
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

Χώρος που δεσμεύεται στην στοίβα σε **κάθε** κλήση της συνάρτησης equalIgnoreCase. Αυτό το κομμάτι μνήμης λέγεται και διάγραμμα ενεργοποίησης (**activation record** ή **stack frame**) της συνάρτησης



Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    → char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

equalIgnoreCase

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

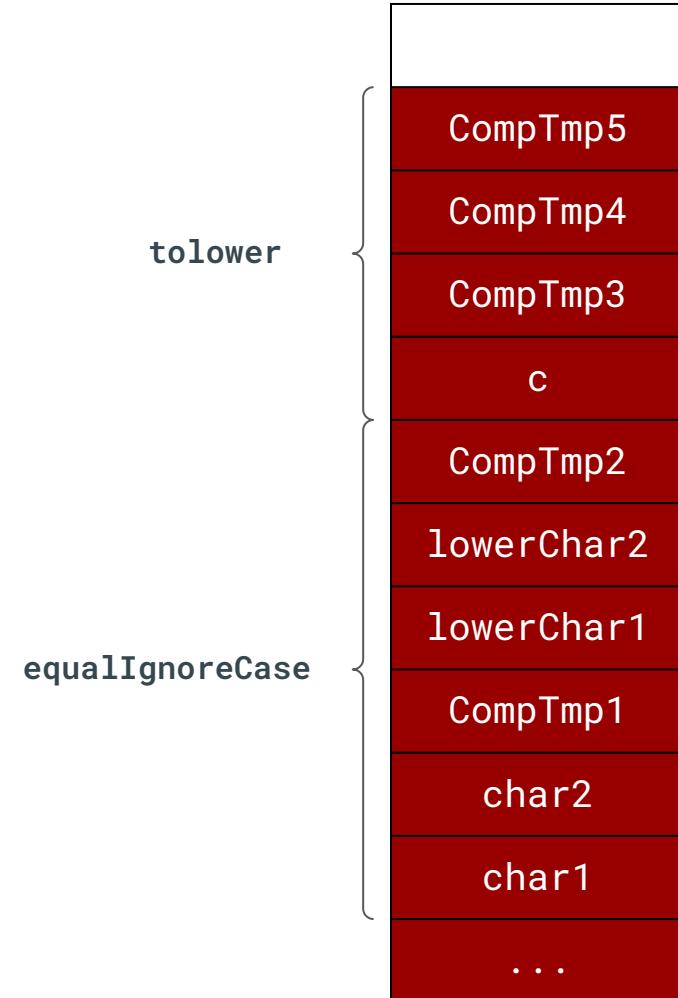
...

Τι θα συμβεί όταν κληθεί η συνάρτηση tolower την πρώτη φορά;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    → return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    → char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

Τι θα συμβεί όταν εκτελεστεί η εντολή return;



Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    → char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

equalIgnoreCase

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

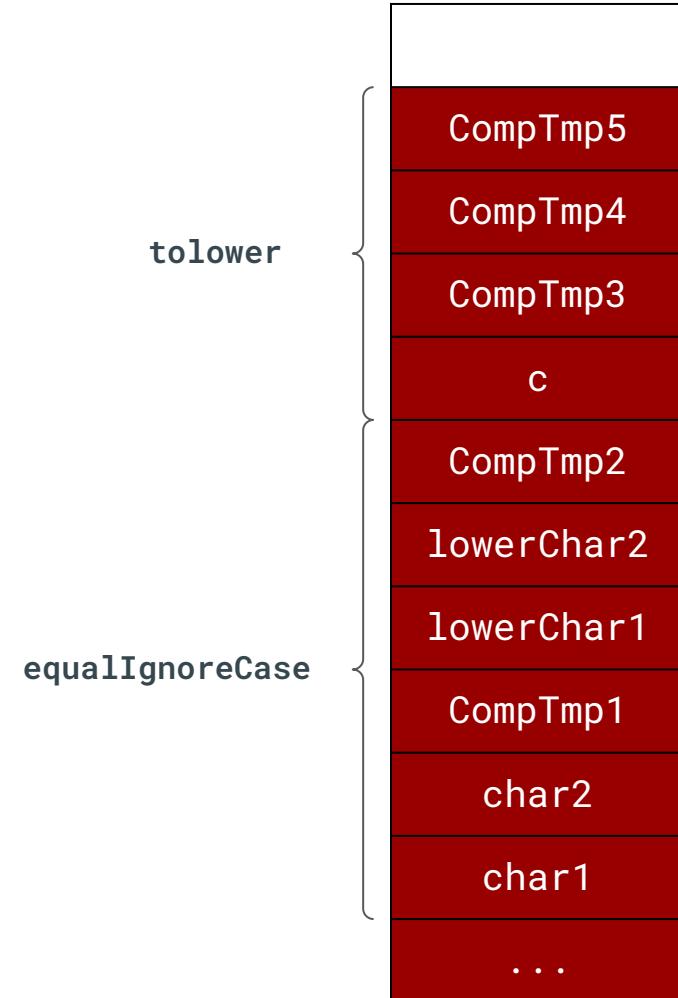
...

Τι θα συμβεί όταν εκτελεστεί η δεύτερη κλήση tolower;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    → return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    → char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

Τι θα συμβεί όταν εκτελεστεί η εντολή return;



Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

equalIgnoreCase

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

Τι θα συμβεί όταν εκτελεστεί η return της equalIgnoreCase;

Τι αποθηκεύεται συνήθως στην στοίβα;

```
char tolower(char c) {  
    if (c >= 'A' && c <= 'Z')  
        return c + ('a' - 'A');  
    return c;  
}  
  
int equalIgnoreCase(char char1, char char2) {  
    char lowerChar1 = tolower(char1);  
    char lowerChar2 = tolower(char2);  
    return lowerChar1 == lowerChar2;  
}
```

Τι θα συμβεί όταν εκτελεστεί η `return` της `equalIgnoreCase`;

...

Είχαμε πει "η αναδρομή πρέπει να τελειώνει". Γιατί;

```
void recurse() {  
    recurse();  
}  
  
int main() {  
    recurse();  
    return 0;  
}
```

```
$ gcc -o rec rec.c  
$ ./rec  
Segmentation fault
```

Πόσο μεγάλη μπορεί να γίνει η στοίβα μου;

Στα περισσότερα συστήματα, το μέγεθος της στοίβας είναι περιορισμένο σε μερικά megabytes (MBs) - καθώς υπάρχει η προσδοκία ότι δεν θα έχουμε εμφωλευμένες κλήσεις εκατομμυρίων συναρτήσεων ή με πολύ μεγάλα τοπικά δεδομένα.

Μπορούμε να βρούμε το μέγεθος της στοίβας μας με την εντολή:

```
$ ulimit -s
```

```
8192
```

Μέγιστο μέγεθος
στοίβας σε KB, δηλαδή
8MB

Tι θα κάνει το ακόλουθο πρόγραμμα;

```
#include <stdio.h>

int main() {
    char bomb[900000];
    printf("Hello World\n");
    return 0;
}
```

Τι θα κάνει το ακόλουθο πρόγραμμα;

```
#include <stdio.h>

int main() {
    char bomb[9000000];
    printf("Hello World\n");
    return 0;
}
```

```
$ gcc -o hello hello.c
$ ulimit -s
8192
$ ./hello
Segmentation fault
$ ulimit -s unlimited
$ ulimit -s
unlimited
$ ./hello
Hello World
```

Είναι γενικά κακή πρακτική
το πρόγραμμά μας να
στηρίζεται σε χρήση
unlimited stack.

Τι κάνουμε όταν η στοίβα δεν είναι αρκετή και
χρειάζεται να χρησιμοποιήσουμε πίνακες μεγάλου
μεγέθους;

Ο Σωρός (Heap)

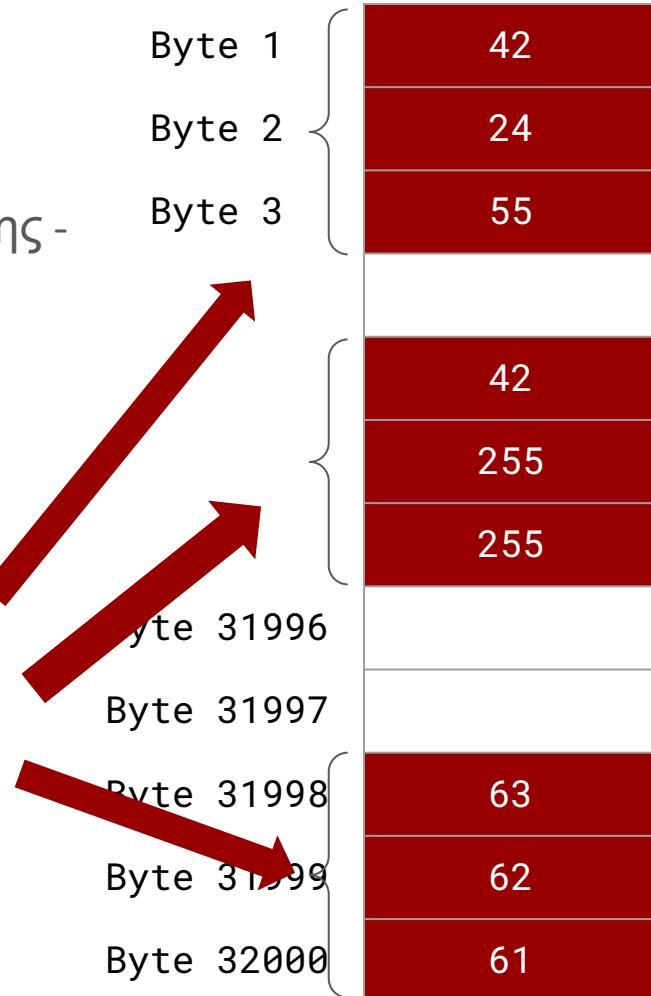
Ο **σωρός (heap)** είναι ένα σύνολο από τοποθεσίες μνήμης - σε τυχαία σειρά. Σε αντίθεση με την στοίβα, ο σωρός μπορεί να δεσμεύσει ολόκληρη την διαθέσιμη μνήμη.



Ο Σωρός (Heap)

Ο **σωρός (heap)** είναι ένα σύνολο από τοποθεσίες μνήμης - σε τυχαία σειρά. Σε αντίθεση με την στοίβα, ο σωρός μπορεί να δεσμεύσει ολόκληρη την διαθέσιμη μνήμη.

Περιοχές της μνήμης δεσμευμένες από τον σωρό (heap)



Δέσμευση Μνήμης Σωρού με την συνάρτηση malloc

Με την βοήθεια της συνάρτησης malloc, μπορούμε να δεσμεύσουμε **δυναμικά** μνήμη στον σωρό. Παράδειγμα για να δημιουργήσουμε έναν πίνακα ακεραίων:

```
int * array = malloc(4 * sizeof(int));
```

stdlib.h

Η μνήμη μετά την εκτέλεση της malloc θα δείχνει ως εξής:

Δημιουργία
πίνακα 4 ακεραίων

int * array



array[0]

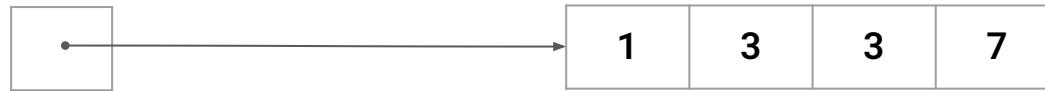
array[3]

Δέσμευση Μνήμης σωρού με malloc

```
int * array = malloc(4 * sizeof(int));
```

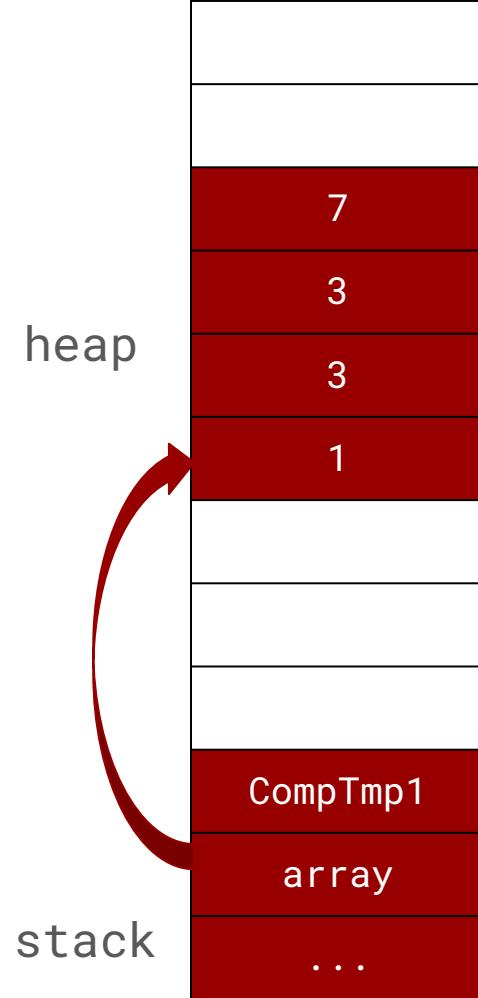
Η μνήμη μετά την εκτέλεση της malloc θα δείχνει ως εξής:

```
int * array
```



array[0]

array[3]



Ο τύπος void

Ο τύπος `void` (που σημαίνει "κενό") είναι ο τύπος της C που χρησιμοποιείται για να δηλώσει το κενό σύνολο - έναν τύπο που δεν μπορεί να έχει στοιχεία. Για παράδειγμα, μια συνάρτηση με τύπο επιστροφής `void` δεν επιστρέφει καμία τιμή.

```
void hi() {  
    printf("Hello World\n");  
    return;  
}
```

Δηλώσεις μεταβλητών με τύπο `void` - π.χ.,
`void a;` δεν είναι επιτρεπτές στην C

Ο τύπος void *

Ο τύπος void * (που σημαίνει δείκτης σε "κενό") είναι ο τύπος της C που χρησιμοποιείται για να δηλώσει έναν δείκτη σε "κάτι" ή αλλιώς μια διεύθυνση. Συνήθως χρησιμοποιείται για να επιστρέψει έναν δείκτη σε μνήμη που μπορεί να χρησιμοποιηθεί ως πίνακας μετά από type casting.

```
void *malloc(size_t size);
```

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

Όμοια με την malloc, απλά μηδενίζει όλα τα στοιχεία της μνήμης πριν επιστρέψει τον δείκτη

Οι συναρτήσεις malloc/calloc επιστρέφουν void * καθώς το αποτέλεσμά τους μπορεί να χρησιμοποιηθεί ως δείκτης σε int, double, char - ανάλογα με την χρήση. Όλα είναι δείκτες ίδιου μεγέθους.

Tι θα κάνει το ακόλουθο πρόγραμμα;

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    char *bomb = malloc(sizeof(char) * 9000000);

    printf("Hello World\n");

    return 0;
}
```

```
$ ./heap
Hello World
```

Tι θα κάνει το ακόλουθο πρόγραμμα;

```
#include <stdio.h>

#include <stdlib.h>

int main() {
    char *bomb = malloc(sizeof(char) * 9000000000L);
    printf("Hello World\n");
    return 0;
}
```

```
$ ./heap
Hello World
```

Tι θα κάνει το ακόλουθο πρόγραμμα;

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char *bomb = malloc(sizeof(char) * 9000000000L);
    bomb[0] = 'A';
    printf("Hello World\n");
    return 0;
}
```

```
$ ./heap
Segmentation fault
```

Προσοχή: Ελέγχουμε ΠΑΝΤΑ το αποτέλεσμα της malloc

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

int main() {
    char *bomb = malloc(sizeof(char) * 90000000000L);
    if (!bomb) {
        perror("bomb is not valid");
        exit(1);
    }
    bomb[0] = 'a';
    printf("Hello World\n");
    return 0;
}
```



Αν δεν υπάρχει αρκετή μνήμη, η malloc μας επιτρέπει έναν NULL δείκτη.
Ελέγχουμε πάντα την τιμή επιστροφής ώστε να χειριστούμε τέτοια σφάλματα

Μάθαμε να δεσμεύουμε μνήμη με την `malloc` -
μας λείπει κάτι;

Απελευθέρωση μνήμης σωρού με την `free`

Η μνήμη που δεσμεύτηκε με την χρήση `malloc/calloc` μπορεί να απελευθερωθεί με την χρήση της συνάρτησης `free` (πάλι από την `stdlib.h`).

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

Ως μόνο όρισμα απαιτεί τον δείκτη στην μνήμη που δεσμεύτηκε αρχικά

Εάν δεν απελευθερώσουμε την μνήμη που δεσμεύσαμε τότε έχουμε **διαρροή μνήμης / memory leak**

Αποδέσμευση Μνήμης με free

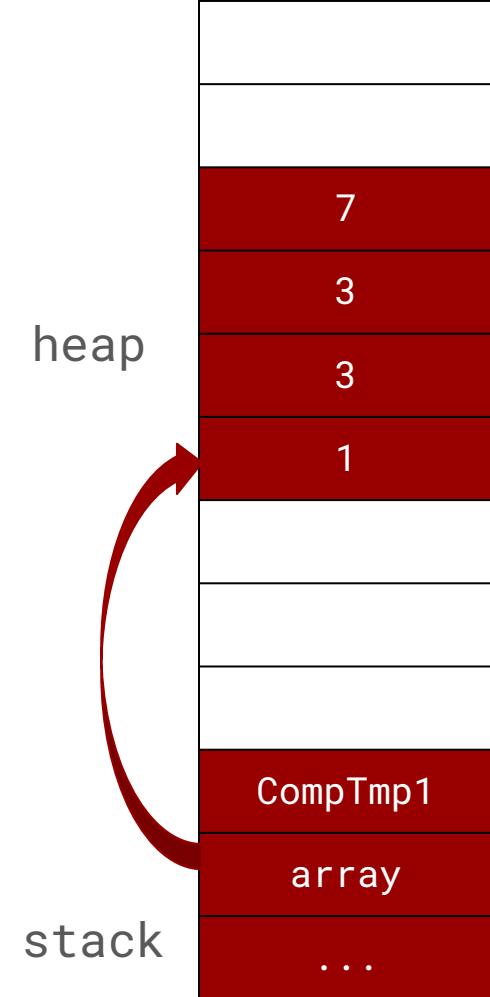
```
int * array = malloc(4 * sizeof(int));  
  
if (!array) {  
    // fail gracefully  
}  
  
free(array);
```

Η ισοδύναμα

array == NULL

Φροντίζουμε κάθε
κλήση malloc να
συνοδεύεται από
μια free

Τι θα συμβεί στην μνήμη μετά την free;

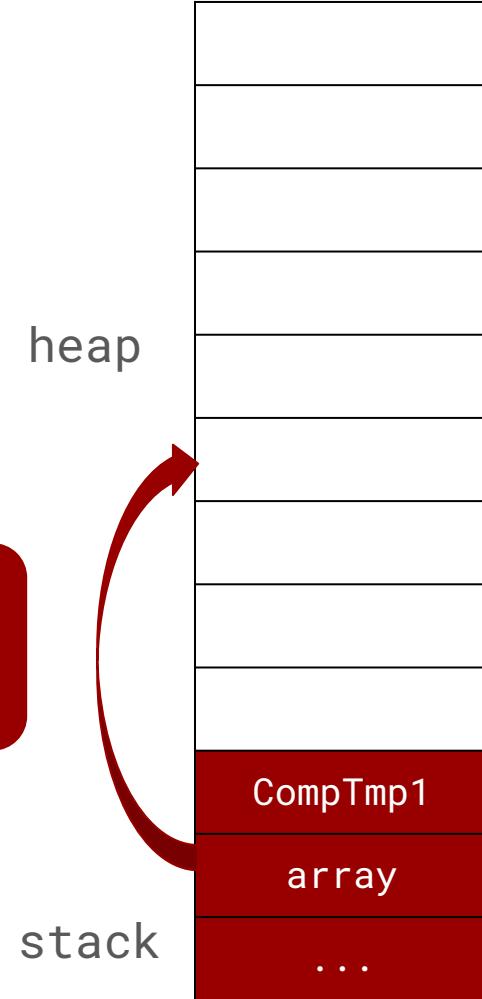


Αποδέσμευση Μνήμης με free

```
int * array = malloc(4 * sizeof(int));  
  
if (!array) {  
    // fail gracefully  
}  
  
free(array);  
  
array[0] = 4;
```

Απαγορεύεται!! Στην καλύτερη περίπτωση το πρόγραμμα θα κρασάρει, στην χειρότερη μπορεί να έχουμε security vulnerability

Τι θα συμβεί αν προσπαθήσω να προσπελάσω μνήμη που έχω αποδεσμεύσει;

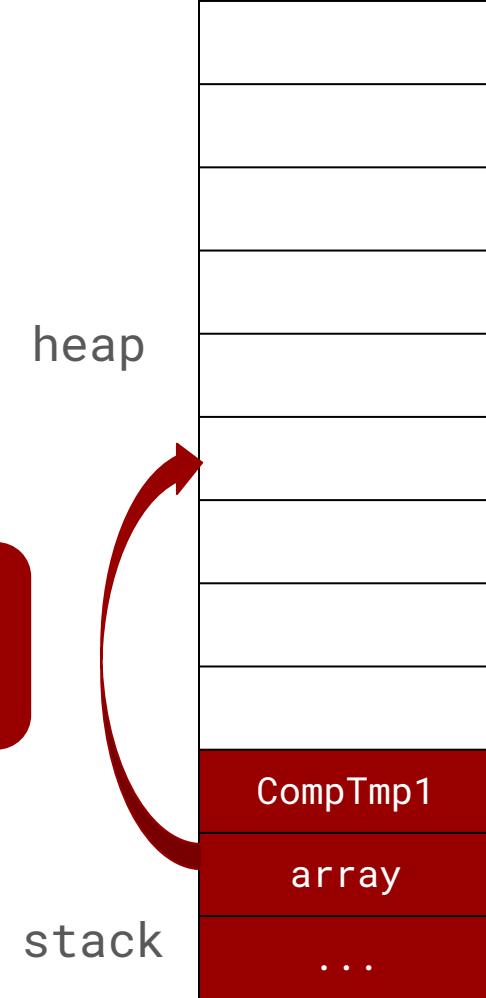


Αποδέσμευση Μνήμης με free

```
int * array = malloc(4 * sizeof(int));  
  
if (!array) {  
    // fail gracefully  
}  
  
free(array);  
  
free(array);
```

Απαγορεύεται!! Στην καλύτερη περίπτωση το πρόγραμμα θα κρασάρει, στην χειρότερη μπορεί να έχουμε security vulnerability

Τι θα συμβεί αν προσπαθήσω να αποδέσμεύσω μνήμη που έχω αποδέσμεύσει;



Αλλαγή μεγέθους με την συνάρτηση realloc

Καθώς τρέχει το πρόγραμμά μας μπορεί να χρειαστούμε περισσότερη (ή λιγότερη!) μνήμη. Με την βοήθεια της realloc, μπορούμε να προσπαθήσουμε να αλλάξουμε το μέγεθος ενός πίνακα. Για παράδειγμα:

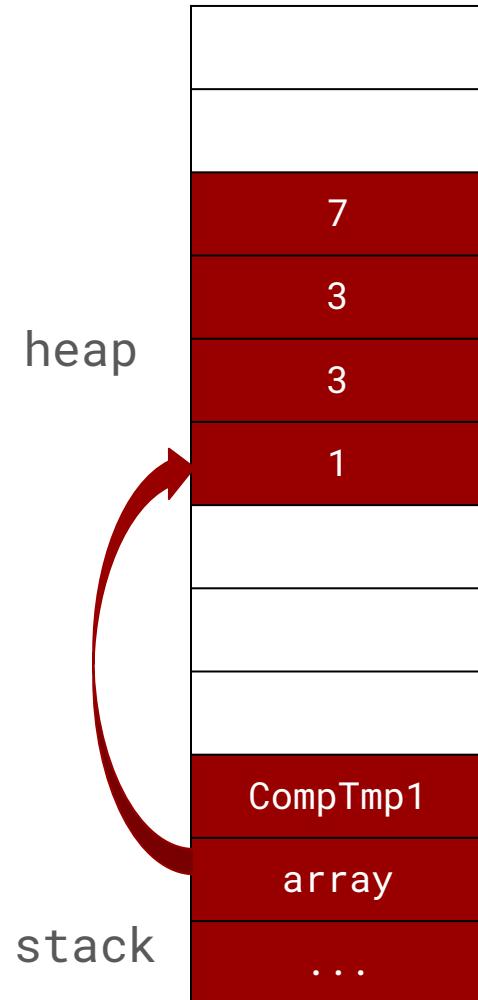
```
int * array = malloc(4 * sizeof(int));  
if (!array) {...}  
  
array = realloc(array, 8192 * sizeof(int));  
if (!array) {...}  
  
array[8000] = 42;  
free(array);
```

Μεγεθύνουμε τον πίνακα από
4 -> 8192 ακεραίους

Προσοχή: ο ίδιος έλεγχος
ισχύει, ΠΑΝΤΑ ελέγχουμε για
NULL αποτέλεσμα

Αλλαγή μεγέθους με την realloc

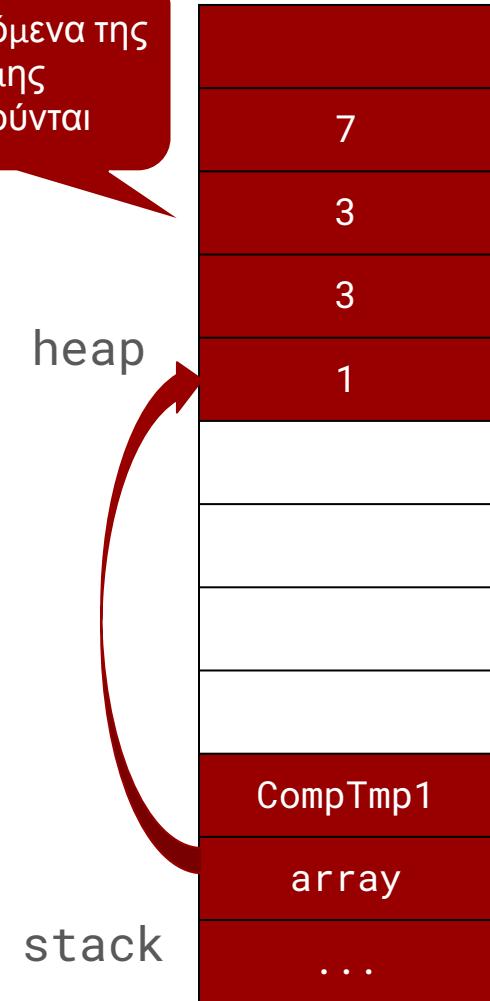
```
int * array = malloc(4 * sizeof(int));  
if (!array) {...}  
array = realloc(array, 8192 * sizeof(int));  
if (!array) {...}  
array[8000] = 42;  
free(array);
```



Αλλαγή μεγέθους με την realloc

Τα περιεχόμενα της
μνήμης
διατηρούνται

```
int * array = malloc(4 * sizeof(int));  
  
if (!array) {...}  
  
array = realloc(array, 8192 * sizeof(int));  
  
if (!array) {...}  
  
array[8000] = 42;  
  
free(array);
```



Θέλω να δημιουργήσω έναν πίνακα 5x5 - αυτή η υλοποίηση είναι:

```
void bar() {  
    int ** array = malloc(5 * sizeof(int*))  
    int i;  
    for(i = 0 ; i < 5 ; i++)  
        array[i] = malloc(5 * sizeof(int));  
    // process loop here  
}
```

Θέλω να δημιουργήσω έναν πίνακα 5x5 - αυτή η υλοποίηση είναι:

```
void bar() {
    int ** array = malloc(5 * sizeof(int *));
    if (!array) {
        return;
    }
    int i;
    for(i = 0 ; i < 5 ; i++) {
        array[i] = malloc(5 * sizeof(int));
        if (!array[i]) {
            return;
        }
    }
    // process loop here
}
```

Θέλω να δημιουργήσω έναν πίνακα 5x5 - αυτή η υλοποίηση είναι:

```
void bar() {
    int ** array = malloc(5 * sizeof(int *));
    if (!array) {
        return;
    }
    int i;
    for(i = 0 ; i < 5 ; i++) {
        array[i] = malloc(5 * sizeof(int));
        if (!array[i]) {
            return;
        }
    }
    // process loop here
    for(i = 0 ; i < 5 ; i++) {
        free(array[i]);
    }
    free(array);
}
```

Διάλεξη 14 - Εμβέλεια, Μνήμη και Συμβολοσειρές

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Εμβέλεια Μεταβλητής (Variable Scope)

Εμβέλεια μιας μεταβλητής λέγεται το μέρος του προγράμματος που η μεταβλητή είναι ορατή / προσβάσιμη.

Το σημείο δήλωσης μιας μεταβλητής στο πρόγραμμα καθορίζει την εμβέλειά της.

Δύο είδη μεταβλητών και εμβέλειας:

1. **Τοπικές** μεταβλητές (local variables)
2. **Παγκόσμιες** μεταβλητές (global variables)

Τοπικές Μεταβλητές (Local Variables)

Μια **τοπική μεταβλητή** (local variable) δηλώνεται μέσα στο σώμα μιας συνάρτησης. Η **εμβέλειά** της είναι από το σημείο δήλωσής της μέχρι το τέλος του block εντολών μέσα στο οποίο ορίστηκε. Αποθηκεύονται στην στοίβα.

```
void foo(int bar) {  
  
    printf("%d\n", bar);  
  
    int i = 42;  
  
    printf("%d\n", i);  
  
    int j;  
  
    for(j = 0; j < i; j++)  
  
        printf("%d %d\n", bar, i * j);  
}
```

Scope της μεταβλητής
ορίσματος bar
(αρχικοποιείται κατά την
κλήση της foo)

Τοπικές Μεταβλητές (Local Variables)

Μια **τοπική μεταβλητή** (local variable) δηλώνεται μέσα στο σώμα μιας συνάρτησης. Η **εμβέλειά** της είναι από το σημείο δήλωσής της μέχρι το τέλος του block εντολών μέσα στο οποίο ορίστηκε. Αποθηκεύονται στην στοίβα.

```
void foo(int bar) {  
  
    printf("%d\n", bar);  
  
    int i = 42;  
    printf("%d\n", i);  
  
    int j;  
    for(j = 0; j < i; j++)  
  
        printf("%d %d\n", bar, i * j);  
}
```

Scope της τοπικής
μεταβλητής i

Τοπικές Μεταβλητές (Local Variables)

Μια **τοπική μεταβλητή** (local variable) δηλώνεται μέσα στο σώμα μιας συνάρτησης. Η **εμβέλειά** της είναι από το σημείο δήλωσής της μέχρι το τέλος του block εντολών μέσα στο οποίο ορίστηκε. Αποθηκεύονται στην στοίβα.

```
void foo(int bar) {  
  
    printf("%d\n", bar);  
  
    int i = 42;  
  
    printf("%d\n", i);  
  
    int j;  
  
    for(j = 0; j < i; j++)  
  
        printf("%d %d\n", bar, i * j);  
  
}
```

Scope της μεταβλητής
ορίσματος j

Τοπικές Μεταβλητές (Local Variables)

Η τοπική μεταβλητή μιας συνάρτησης δεν είναι ορατή σε κάποια άλλη συνάρτηση και επομένως μπορούμε να δηλώσουμε μεταβλητές με το ίδιο όνομα σε άλλες συναρτήσεις και να αναφέρονται σε άλλες θέσεις μνήμης.

```
void foo() {  
    int i = 43;  
}  
  
int main() {  
    int i = 42;  
    foo();  
    return i;  
}
```

Τι θα επιστρέψει αυτό το πρόγραμμα;
\$ gcc -o local local.c
\$./local
\$ echo \$?
42

Τοπικές Μεταβλητές (Local Variables)

Η τοπική μεταβλητή μιας συνάρτησης δεν είναι ορατή σε κάποια άλλη συνάρτηση και επομένως μπορούμε να δηλώσουμε μεταβλητές με το ίδιο όνομα σε άλλες συναρτήσεις και να αναφέρονται σε άλλες θέσεις μνήμης.

```
void foo(int i) {  
    i++;  
}  
  
int main() {  
    int i = 42;  
    foo(i);  
    return i;  
}
```

Τι θα επιστρέψει αυτό το πρόγραμμα;
\$ gcc -o local2 local2.c
\$./local2
\$ echo \$?
42

Παγκόσμιες Μεταβλητές (Global Variables)

Μια **παγκόσμια μεταβλητή** (global variable) δηλώνεται έξω από οποιαδήποτε συνάρτηση. Η **εμβέλειά** της είναι από το σημείο δήλωσής της μέχρι το τέλος του **αρχείου** μέσα στο οποίο ορίστηκε.

```
int baz = 42;

void foo() {
    printf("baz: %d\n", baz);
}

void bar() {
    baz++;
}
```

Scope της μεταβλητής
ορίσματος baz - ορατή και
στην foo και στην bar

Επισκίαση Μεταβλητής (Variable Shadowing)

Όταν μια δήλωση μεταβλητής βρίσκεται εντός εμβέλειας άλλης μεταβλητής με το ίδιο όνομα, τότε η εσωτερική μεταβλητή επισκιάζει (shadows) την άλλη.

```
#include <stdio.h>

int baz = 42;

int main() {
    int baz = 43;
    printf("baz: %d\n", baz);
    return 0;
}
```

Επισκίαση Μεταβλητής (Variable Shadowing)

Όταν μια δήλωση μεταβλητής βρίσκεται εντός εμβέλειας άλλης μεταβλητής με το ίδιο όνομα, τότε η εσωτερική μεταβλητή επισκιάζει (shadows) την άλλη.

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

```
int baz = 42;  
  
int main() {  
  
    int baz = 43;  
  
    printf("baz: %d\n", baz);  
  
    return 0;  
}
```

Scope της παγκόσμιας
μεταβλητής baz

Scope της τοπικής
μεταβλητής baz

Τι θα τυπώσει;

\$./shadow
baz: 43

Στατικές Μεταβλητές (Static Variables)

Μια μεταβλητή λέγεται **στατική** (static), όταν διατηρεί την τιμή της ανάμεσα σε κλήσεις συναρτήσεων μέχρι το τέλος του προγράμματος. Οι στατικές μεταβλητές αρχικοποιούνται μόνο στην πρώτη κλήση της συνάρτησης.

```
#include <stdio.h>

void foo() {

    static int counter; int i = 0; counter = 0;

    counter++; i++;

    printf("%d %d\n", counter, i);
}

int main() {
    foo(); foo(); foo();
    return 0;
}
```

Προσοχή: αν αρχικοποιήσουμε
ΕΚΤΟΣ δήλωσης μεταβλητής τότε
η τιμή δεν διατηρείται

Τι θα τυπώσει;

\$./static

1 1

1 1

1 1

Η Παγκόσμια / Στατική Μνήμη (Global / Static Memory)

Η **παγκόσμια / στατική** μνήμη είναι ένα μέρος της μνήμης του προγράμματος ξεχωριστό από την στοίβα και στον σωρό που αποθηκεύονται παγκόσμιες, στατικές μεταβλητές καθώς και δεδομένα του προγράμματος (ακόμα και ο κώδικας ο ίδιος).

Η μνήμη αυτή αρχικοποιείται όταν ξεκινά το πρόγραμμα και αποδεσμεύεται όταν τερματίσει.



Έχει πολλές υποκατηγορίες (data, code, .bss - άλλο μάθημα)

Παράδειγμα με κάθε μνήμη

```
#include <stdlib.h>

char global[3] = {42, 24, 55};

void foo(char c1, char c2) {
    char c3 = 63;
    char * tmp = malloc(3 * sizeof(char));
    tmp[0] = 42; tmp[1] = 255; tmp[2] = 255;
}

int main() {
    foo(61, 62);
    return 0;
}
```

Παγκόσμια / Στατική

Σωρός

Στοίβα

Byte 1	42
Byte 2	24
Byte 3	55
Byte 1	42
Byte 2	255
Byte 3	255
Byte 31996	
Byte 31997	
Byte 31998	63
Byte 31999	62
Byte 32000	61

Τα περιεχόμενα της στατικής μνήμης

Κάθε αρχείο C μπορεί να ορίζει τις δικές του παγκόσμιες μεταβλητές και τα ονόματά τους σώζονται ως σύμβολα από τον μεταγλωττιστή.

```
int global_buffer[100000000] = {1};  
  
void foo() {  
  
    static int counter = 0;  
  
    counter++;  
  
}  
  
int main() {  
  
    foo();  
  
    return 0;  
}
```

Αν η μεταβλητή αρχικοποιηθεί με μη-μηδενικές τιμές, ο χώρος που χρειάζεται και όλες οι σχετικές τιμές αποθηκεύονται στο αρχείο από τον μεταγλωττιστή

```
$ gcc -c memory.c  
$ nm memory.o  
0000000000000000 b counter.0  
0000000000000000 T foo  
0000000000000000 D global_buffer  
0000000000000016 T main  
$ du -sh memory.o  
382M      memory.o
```

Τα περιεχόμενα της στατικής μνήμης

Συμβολοσειρές που δεν χρησιμοποιούνται για την αρχικοποίηση ενός τοπικού πίνακα αποθηκεύονται επίσης στην στατική μνήμη.

```
#include <stdio.h>

int main() {
    char str1[] = "Hello World", str2[] = "Hello World";
    printf("%p %p\n", str1, str2);
    return 0;
}
```

\$ gcc -o const const.c
\$./const
0x7ffc2576ccc4 0x7ffc2576ccb8

Συναρτήσεις βιβλιοθήκης για συμβολοσειρές:
string.h

Πίνακας Χαρακτήρων / Συμβολοσειρά (String)

Ένας πίνακας από χαρακτήρες λέγεται και **αλφαριθμητικό / συμβολοσειρά (string)**. Λόγω της συχνής χρήσης τους, έχουμε αρκετές συντομεύσεις για αυτούς. Οι τρεις παρακάτω δηλώσεις είναι ισοδύναμες:

```
char hello[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\n', '\0'};  
char hello[] = {72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 10, 0};  
char hello[] = "Hello World\n";
```

Προσοχή: τα string
τερματίζονται πάντα με
το null byte

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\n'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------

Η συνάρτηση strlen

Η συνάρτηση `strlen` επιστρέφει τον αριθμό των χαρακτήρων μιας συμβολοσειράς χωρίς να μετράει το null byte ('\0').

```
#include <stdio.h>
#include <string.h>
int main(int argc, char ** argv) {
    if (argc != 2) return 1;
    printf("Arg1 length: %d\n", strlen(argv[1]));
    return 0;
}
```

```
$ gcc -o strlen strlen.c
$ ./strlen "hello world"
Arg1 length: 11
```

Η συνάρτηση `strlen`

Μια πιθανή υλοποίηση:

```
size_t strlen(char * str) {  
    size_t length = 0;  
    while(*str++) length++;  
    return length;  
}
```

Η συνάρτηση strcmp

Η συνάρτηση strcmp ελέγχει αν δύο συμβολοσειρές A και B είναι ίσες. Αν είναι επιστρέφει 0, αλλιώς επιστρέφει θετική τιμή αν το A > B (αρνητική αν B > A)

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

int main(int argc, char ** argv) {
    if (argc != 3) return 1;
    printf("strcmp(arg1, arg2): %d\n",
           strcmp(argv[1], argv[2]));
    return 0;
}
```

```
$ ./strcmp foo bar
strcmp(arg1, arg2): 4
$ ./strcmp foo foo
strcmp(arg1, arg2): 0
$ ./strcmp f bar
strcmp(arg1, arg2): 4
$ ./strcmp bar f
strcmp(arg1, arg2): -4
$ ./strcmp bar c
strcmp(arg1, arg2): -1
$ ./strcmp bar ba
strcmp(arg1, arg2): 114
```

Η συνάρτηση strcmp

Μια πιθανή υλοποίηση:

```
int strcmp(char * str1, char * str2) {  
    while(*str1 && (*str1 == *str2)) {  
        str1++;  
        str2++;  
    }  
    return *str1 - *str2;  
}
```

Η συνάρτηση strcpy

Η συνάρτηση strcpy αντιγράφει (copy) μια συμβολοσειρά (2o όρισμα) σε μια άλλη (1o όρισμα).

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

int main(int argc, char ** argv) {
    char hello[16] = "Hello!";
    char world[16];
    strcpy(world, hello);
    printf("world: %s\n", world);
    return 0;
}
```

```
$ ./strcpy
world: Hello!
```

Η συνάρτηση strcpy είναι **ιδιαίτερα επικίνδυνη** (ασφάλεια!) καθώς γράφει στην μνήμη χωρίς να ελέγχει αν υπάρχει επαρκής χώρος - πρέπει να ελέγξουμε εμείς. Ελαχιστοποιούμε την χρήση της (man strcpy για μια λίγο καλύτερη - αλλά πάλι κακή - εκδοχή).

Η συνάρτηση strcpy

Μια πιθανή υλοποίηση:

```
char * strcpy(char * dst, char * src) {  
    char * original_dst = dst;  
  
    while(*src) *dst++ = *src++;  
  
    *dst = '\0';  
  
    return original_dst;  
}
```

Η συνάρτηση strcat

Η συνάρτηση strcat συνενώνει (concatenates) δύο συμβολοσειρές στην πρώτη ξεκινώντας την αντιγραφή του 2ου ορίσματος από το τέλος (null byte) της πρώτης συμβολοσειράς.

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

int main(int argc, char ** argv) {
    char hello[16] = "Hello ";
    char world[16] = "World!";
    strcat(hello, world);
    printf("concat: %s\n", hello);
    return 0;
}
```

```
$ ./strcat
concat: Hello World!
```

Προσοχή: η strcat έχει ακριβώς τα ίδια προβλήματα με την strcpy. Ελέγχουμε πάντα ότι υπάρχει αρκετός χώρος στον πίνακα για την αντιγραφή.

Η συνάρτηση strcat

Μια πιθανή υλοποίηση:

```
char *strcat(char *dst, char *src) {  
    char *original_dst = dst;  
  
    while (*dst) dst++;  
  
    while (*src) *dst++ = *src++;  
  
    *dst = '\0';  
  
    return original_dst;  
}
```

Διάλεξη 15 - Επίλυση Προβλημάτων

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Έχω έναν πίνακα ακεραίων και όλα τα στοιχεία του είναι διπλά εκτός από ένα. Πως το βρίσκω αποδοτικά;

Θέλω μια συνάρτηση που να δέχεται έναν πίνακα 100 ακεραίων και να επιστρέψει τον μέσο όρο. Πως;

```
int average(int grades[100]) {  
    int i, sum = 0;  
    for(i = 0; i < 100; i++) {  
        sum += grades[i];  
    }  
    return sum / 100;  
}
```

Θέλω μια συνάρτηση που να δέχεται έναν πίνακα 100 ακεραίων και έναν ακέραιο και να γυρνάει την θέση του στοιχείου αν το βρήκε ή -1. Πως;

```
int find(int haystack[100], int needle) {  
    int i;  
    for(i = 0; i < 100; i++) {  
        if (haystack[i] == needle) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Θέλω μια συνάρτηση atoi που να παίρνει ένα πίνακα χαρακτήρων (μόνο ψηφία) και να επιστρέψει έναν ακέραιο. Πως;

```
int atoi(char digits[]) {  
    int result = 0;  
  
    for(int i = 0; digits[i]; i++) {  
  
        result = 10 * result + digits[i] - '0';  
  
    }  
  
    return result;  
}
```

Τι μπορεί να πάει στραβά με αυτήν την συνάρτηση;

Θέλω να βρω το άθροισμα των τέλειων τετραγώνων μέσα σε ένα διάστημα αριθμών. Πως το βρίσκω αποδοτικά;

Ποια η διαφορά ενός `char * array[]` και ενός `char ** array`; Έχει διαφορά από έναν `char array[10][10]`;

Θέλω να γράψω μια συνάρτηση get_two_chars που να διαβάζει *δύο* χαρακτήρες και να τους επιστρέψει. Πως;

Θέλω μια συνάρτηση που να αντιστρέψει τα ψηφία ενός αριθμού,
για παράδειγμα αν δοθεί το 123, θέλω να επιστρέψει το 321.
Πως;

Θέλω να βρω αν ένας αριθμός είναι παλινδρομικός. Πως μπορώ να το βρώ;

Γράψτε μια συνάρτηση swap που ανταλλάζει δύο ακεραίους

```
#include <stdio.h>

int main() {
    int a = 100, b = 200;
    printf("%d %d\n", a, b);
    swap( ... );
    printf("%d %d\n", a, b);
    return 0;
}
```

Γράψτε μια συνάρτηση swap που ανταλλάζει δύο ακεραίους

```
#include <stdio.h>

void swap(int *a, int *b) {

    int tmp = *a;

    *a = *b;

    *b = tmp;

}

int main() {

    int a = 100, b = 200;

    printf("%d %d\n", a, b);

    swap(&a, &b);

    printf("%d %d\n", a, b);

    return 0;

}
```

Θέλω να τυπώσω "yes" αν ένα στοιχείο βρίσκεται σε έναν δισδιάστατο πίνακα. Πως;

Θέλω να δημιουργήσω έναν δισδιάστατο πίνακα στον σωρό. Πως μπορώ να το κάνω αυτό;

Θέλω μια αναδρομική υλοποίηση που να υπολογίζει τους fibonacci αριθμούς αποδοτικά. Γίνεται;

Διάλεξη 16 - Πολυπλοκότητα και Προεπεξεργαστής

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Πολυπλοκότητα (Complexity)

Η **πολυπλοκότητα** είναι ένα μέτρο εκτίμησης της απόδοσης αλγορίθμων ως συνάρτηση του μεγέθους του προβλήματος που επιλύουν. Δύο μετρικές:

1. Χρόνος εκτέλεσης
2. Χώρος μνήμης που απαιτείται

Παράδειγμα: έστω η το μέγεθος του προβλήματος. Θέλουμε να μπορούμε να εκφράσουμε τον χρόνο εκτέλεσης του αλγορίθμου μας: $t = f(n)$.

Πολυπλοκότητα και Big-O Notation

- Ορισμός κλάσεων πολυπλοκότητας

a. Άνω όριο $g = O(f) \Leftrightarrow \exists c. \exists n_0. \forall n > n_0. g(n) < c \cdot f(n)$

b. Κάτω όριο $g = \Omega(f) \Leftrightarrow \exists c. \exists n_0. \forall n > n_0. g(n) > c \cdot f(n)$

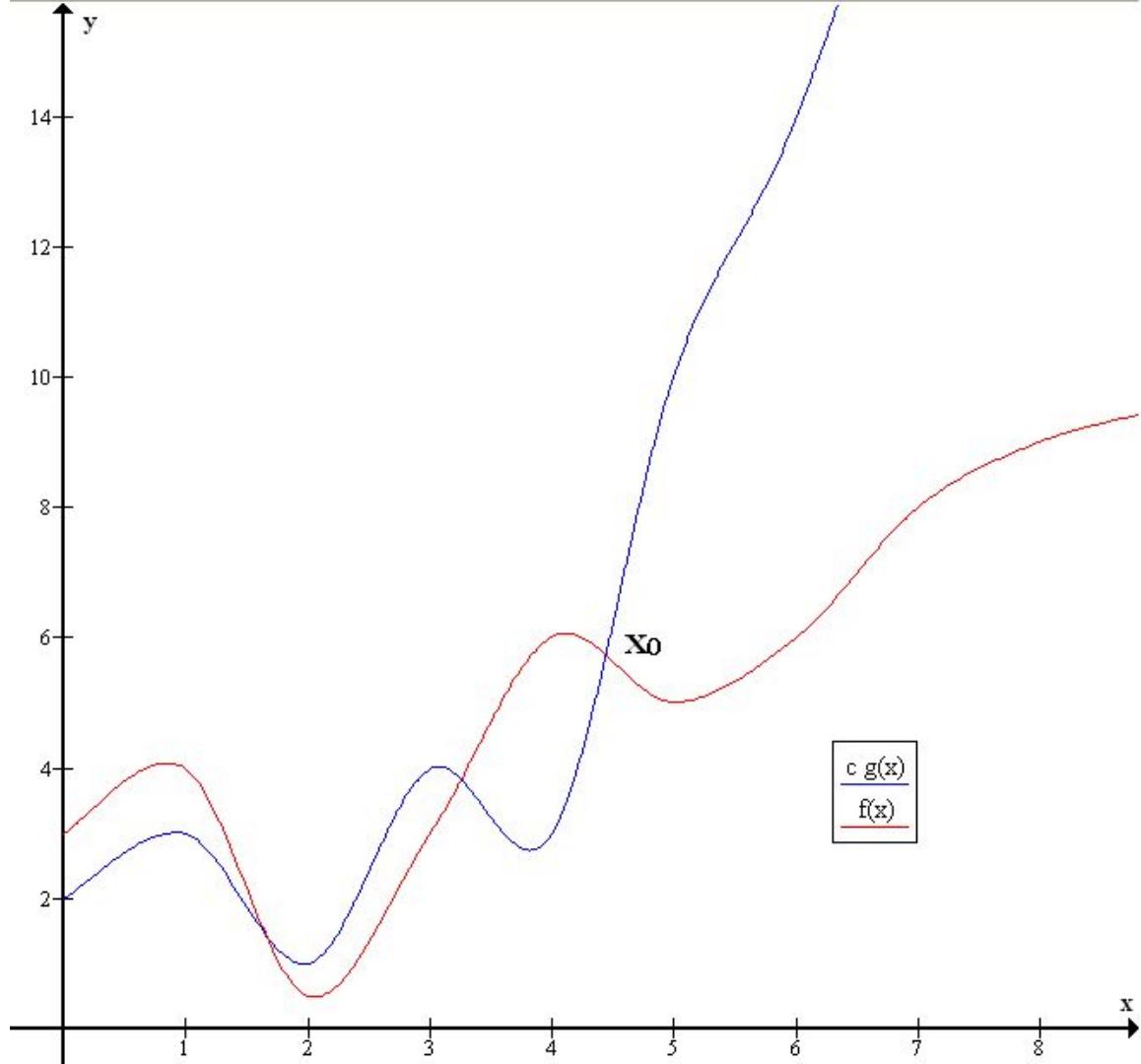
c. Τάξη μεγέθους

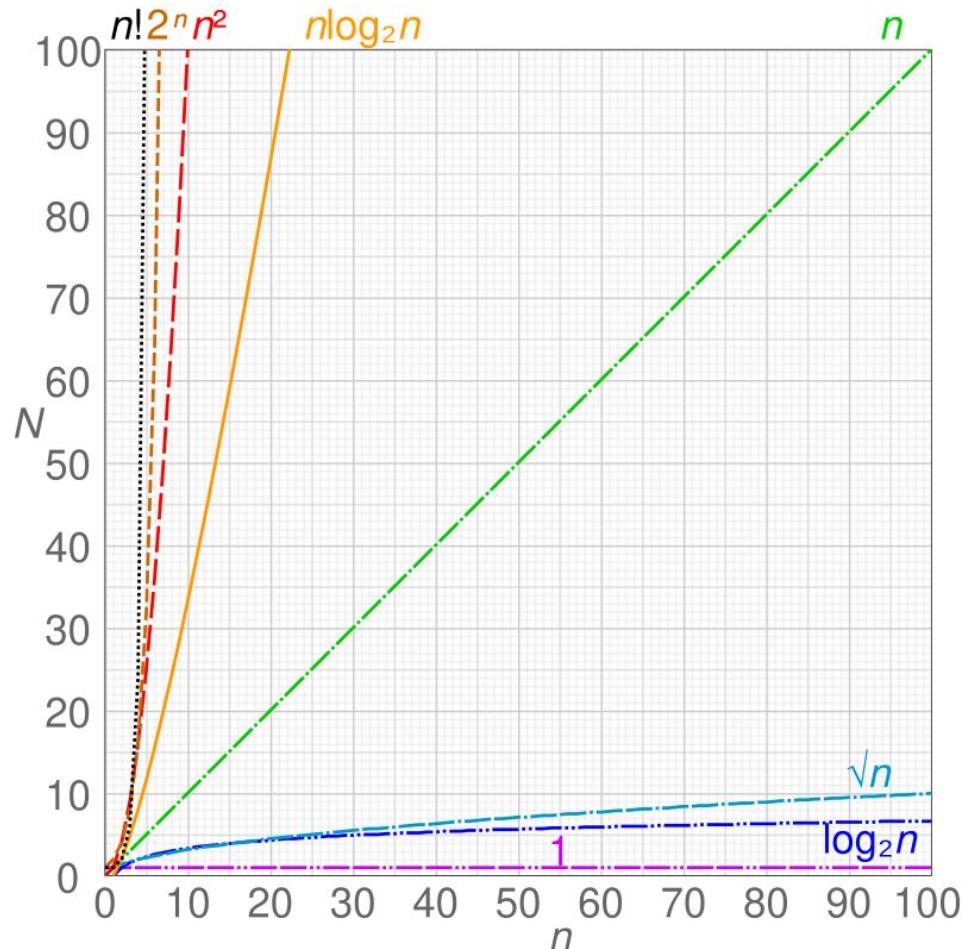
$$g = \Theta(f) \Leftrightarrow \exists c_1, c_2. \exists n_0. \forall n > n_0. c_1 \cdot f(n) < g(n) < c_2 \cdot f(n)$$

- Διάταξη μερικών κλάσεων πολυπλοκότητας

$$\begin{aligned} O(1) &< O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) \\ &< O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n) \end{aligned}$$

$$f(x) = 0(g(x))$$





Θέλω να το γινόμενο όλων των περιττών από το 1 μέχρι το N που διαιρούνται με το 7. Πως;

Ένα for loop με μια μεταβλητή που αυξάνεται και έλεγχο για $\text{mod } 7 = 0$:

```
for(product = 1, i = 1; i < N ; i += 2) {  
    if ( i % 7 == 0 )  
        product *= i;  
}
```

Χρόνος: $O(N)$
Χώρος: $O(1)$

Θέλω μια συνάρτηση atoi που να παίρνει ένα πίνακα χαρακτήρων (μόνο ψηφία) και να επιστρέψει έναν ακέραιο. Πως;

```
int atoi(char digits[]) {  
    int result = 0;  
  
    for(int i = 0; digits[i]; i++) {  
  
        result = 10 * result + digits[i] - '0';  
  
    }  
  
    return result;  
}
```

Χρόνος: $O(N)$
Χώρος: $O(1)$

Χρήση της συνάρτησης getchar()

Διαδοχικές κλήσεις της getchar() διαβάζουν διαδοχικούς χαρακτήρες. Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    int ch, sum = 0;
    printf("Enter characters: ");
    while( (ch = getchar()) != '\n' && ch != EOF ) {
        printf("%c", ch);
        sum++;
    }
    printf("\nTotal characters: %d\n", sum);
    return 0;
}
```

Χρόνος: $O(N)$
Χώρος: $O(1)$

Δυναμικοί Πίνακες με την συνάρτηση malloc()

Με την βοήθεια των δεικτών, μπορούμε να χρησιμοποιήσουμε **δυναμικούς** πίνακες, πίνακες των οποίων το μέγεθος καθορίζεται **δυναμικά**, δηλαδή την στιγμή που τρέχει το πρόγραμμα. Παράδειγμα:

```
int * array = malloc(N * sizeof(int));  
  
for(int i = 0 ; i < N ; i++)  
  
    array[i] = i * i;
```

Χρόνος: $O(N)$
Χώρος: $O(N)$

Το γνωστό μας Παράδειγμα: Υπολογισμός Βαθμολογίας

```
// Compute grades using the class formula
int grade(int final_exam, int homework, int lab, int year) {
    if (year <= 1) {
        return final_exam * 50 / 100 + homework * 30 / 100 + lab * 20 / 100;
    } else {
        return final_exam * 70 / 100 + homework * 30 / 100;
    }
}
```

Χρόνος: O(1)
Χώρος: O(1)

Εύρεση Μέγιστου Στοιχείου σε Πίνακα N x N

```
int find_max(int **matrix, size_t n) {  
    int i, j;  
    for(i = 0; i < n; i++) {  
        for(j = 0; j < n; j++) {  
            if (matrix[i][j] > max) max = matrix[i][j];  
        }  
    }  
    return max;  
}
```

Χρόνος: $O(n^2)$
Χώρος: $O(1)$

Η συνάρτηση παραγοντικό (factorial)

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

Χρόνος: $O(n)$
Χώρος: $O(n)$

Η συνάρτηση fibonacci

```
int fib(int n) {  
    if (n == 0 || n == 1) return 1;  
    return fib(n - 1) + fib(n - 2);  
}
```

Χρόνος: $O(2^n)$
Χώρος: $O(n)$

Η συνάρτηση `strlen`

Μια πιθανή υλοποίηση:

```
size_t strlen(char * str) {  
    size_t length = 0;  
  
    while(*str++) length++;  
  
    return length;  
}
```

Τι πολυπλοκότητας είναι η συνάρτηση `strlen` ως προς το μέγεθος της συμβολοσειράς;

Χρόνος: $O(n)$

Χώρος: $O(1)$

Η συνάρτηση strcmp

Μια πιθανή υλοποίηση:

```
int strcmp(char * str1, char * str2) {  
    while(*str1 && (*str1 == *str2)) {  
        str1++;  
        str2++;  
    }  
    return *str1 - *str2;  
}
```

Τι πολυπλοκότητας είναι η συνάρτηση strcmp ως προς το μέγεθος των συμβολοσειρών (έστω n και m);

Χρόνος: $O(\min(m, n))$
Χώρος: $O(1)$

Αθροισμα τέλειων τετραγώνων

```
int isPerfectSquare(int n) {  
    int root = sqrt(n);  
    return root * root == n;  
}  
  
int low = atol(argv[1]);  
int high = atol(argv[2]);  
  
int i, sum = 0;  
for(i = low ; i <= high ; i++) {  
    if (isPerfectSquare(i))  
        sum += i;
```

Χρόνος: $O(n)$
Χώρος: $O(1)$

Αθροισμα τέλειων τετραγώνων

```
int low = atoll(argv[1]);  
int high = atoll(argv[2]);  
int i, sum = 0;  
for(i = sqrt(low) ; i <= sqrt(high) ; i++)  
    sum += i*i ;
```

Χρόνος: $O(\sqrt{n})$
Χώρος: $O(1)$

Εύρεση του κατόπτρου ενός ακεραίου

```
int mirror(int n) {  
    int result = 0, tmp;  
    while(n > 0) {  
        tmp = n % 10;  
        result = 10 * result + tmp;  
        n /= 10;  
    }  
    return result;  
}
```

Χρόνος: $O(\log n)$
Χώρος: $O(1)$

Έλεγχος αν ένας αριθμός είναι πρώτος - τι χρονική πολυπλοκότητα έχει;

Μεταγλωττιστές και Προεπεξεργαστές

Μεταγλωτιστές (Compilers)

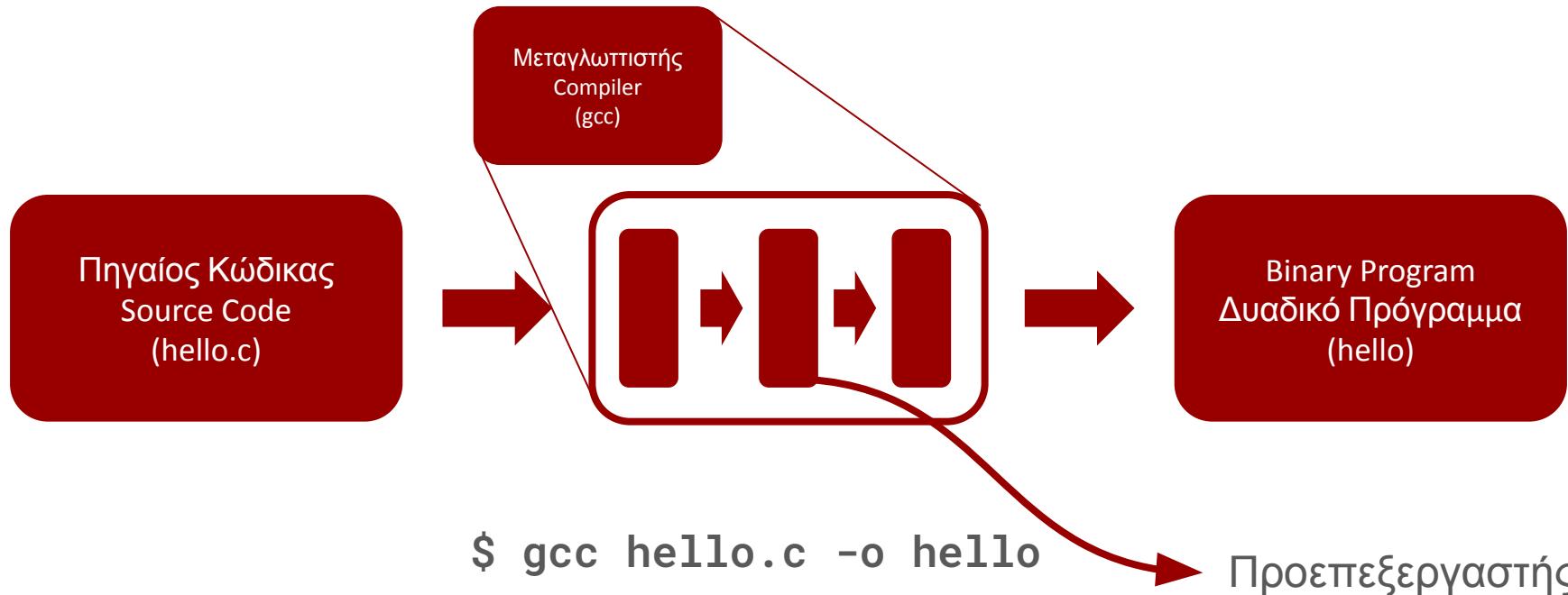
Μεταγλωτιστής (compiler) είναι ένα πρόγραμμα που μετατρέπει εντολές μιας γλώσσας προγραμματισμού σε κώδικα μηχανής ώστε να μπορεί να διαβαστεί και να τρέξει από τον υπολογιστή.



```
$ gcc hello.c -o hello
```

Προεπεξεργαστής (Preprocessor)

Ο προεπεξεργαστής (preprocessor) είναι ένα υποσύστημα του μεταγλωτιστή.



Προεπεξεργαστής (Preprocessor)

Μπορούμε να δούμε την έξοδο του προεπεξεργαστή με το -E (ή τρέχοντας το πρόγραμμα του προεπεξεργαστή το ίδιο - cpp):



\$ gcc -E hello.c -o processed.c

ή

\$ cpp hello.c -o processed.c

Εντολές Προεπεξεργαστή (Preprocessor Directives)

Όλες οι εντολές στον προεπεξεργαστή ξεκινάνε με το σύμβολο #. Ενδεικτικά:

1. **#include**
2. **#define**
3. **#if #else #elif #endif**
4. **#ifdef #ifndef**

Η εντολή #include

Η εντολή `#include <file.h>` εισάγει τα περιεχόμενα του αρχείου `file.h` στο σημείο που γράφτηκε στο πρόγραμμα.

Που βρίσκονται αυτά τα αρχεία; Σε προκαθορισμένους φακέλους στο λειτουργικό σας ή σε φακέλους που προσδιορίζονται με το όρισμα `-I` του `gcc`

Δεύτερη εκδοχή: `#include "file.h"` - σε αυτήν την περίπτωση ο μεταγλωττιστής ψάχνει πρώτα στον φάκελο που βρίσκεται το πηγαίο αρχείο.

Η εντολή #define

Η εντολή `#define` μας επιτρέπει να ορίσουμε μακροεντολές (macros)

1. Ορισμός σταθεράς:

```
#define TRUE 1
```

2. Ορισμός υπολογισμού:

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

Η μακροεντολή αντικαθίσταται στον κώδικα πριν την μεταγλώττιση

Τι θα επιστρέψει το παρακάτω πρόγραμμα;

```
#define PROD 2*5

int main() {
    return 20 / PROD;
}
```

```
$ cpp prod.c
# 0 "prod.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3
# 0 "<command-line>" 2
# 1 "prod.c"
```

```
int main() {
    return 20 / 2*5;
}
$ gcc -o prod prod.c
$ ./prod
$ echo $?
50
```

Η τιμή της μακροεντολής μπορεί να περαστεί από την γραμμή εντολών

Χρησιμοποιούμε την σύνταξη -DMACRO=VALUE όπου MACRO η μακροεντολή και VALUE η τιμή που θέλουμε να δώσουμε. Για παράδειγμα:

```
int main() {  
    return 20 / PROD;  
}  
  
$ gcc -DPROD=10 -o prod prod.c  
$ ./prod  
$ echo $?  
2
```

Η εντολή #if #else #endif

Έχουν μορφή παρόμοια με την δομή ελέγχου if στην C αλλά δρουν στο επίπεδο του κώδικα. Σε τι θα προεπεξεργαστεί το ακόλουθο πρόγραμμα:

```
int main() {  
    #if 0  
        return 42;  
    #else  
        return 1;  
    #endif  
}
```

Η εντολή #if #else #endif

Έχουν μορφή παρόμοια με την δομή ελέγχου if στην C αλλά δρουν στο επίπεδο του κώδικα. Σε τι θα προεπεξεργαστεί το ακόλουθο πρόγραμμα:

```
int main() {  
#if 0  
    return 42;  
#else  
    return 1;  
#endif  
}  
  
$ gcc -E example.c  
# 0 "example.c"  
# 0 "<built-in>"  
# 0 "<command-line>"  
# 1 "/usr/include/stdc-predef.h" 1 3 4  
# 0 "<command-line>" 2  
# 1 "example.c"  
int main() {  
    return 1;  
}
```

Η εντολή #ifdef #ifndef

Με τις εντολές #ifdef / #ifndef μπορούμε να ελέγξουμε αν μια μακροεντολή έχει οριστεί ή όχι:

```
#define DEBUG

#ifndef DEBUG
    printf("debugging is on\n");
#else
    printf("debugging is off\n");
#endif

#ifndef DEBUG
    printf("optimizations are on\n");
#endif
```

Διάλεξη 17 - Δυαδική Αναζήτηση και Ταξινόμηση

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

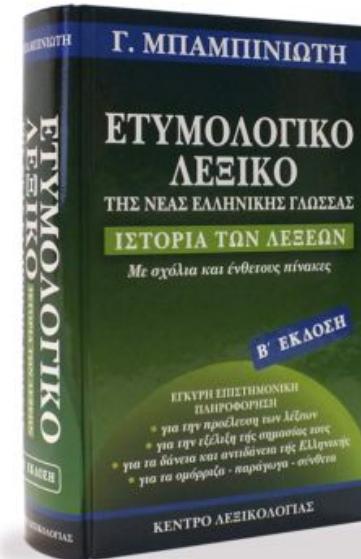
Έχω έναν αριθμό στο νου μου στο διάστημα $[1, 10^9]$

Θέλετε να τον βρείτε. Μπορείτε να με ρωτήσετε ότι ερώτηση θέλετε και εγώ απαντάω **ναι** ή **όχι**.

Τι θα με ρωτήσετε; Πόσες προσπάθειες χρειάζεστε για να βρείτε τον αριθμό μου;

Αναζήτηση Λήμματος σε Λεξικό

Αναζητώ την ετυμολογία μιας λέξης,
δεν διατρέχω όλο το λεξικό μέχρι να την βρω



Guess Who

Στόχος: βρες ποιον χαρακτήρα
έχω επιλέξι με τις λιγότερες
ερωτήσεις



Θέλω μια συνάρτηση που να δέχεται έναν πίνακα 100 ακεραίων και έναν ακέραιο και να γυρνάει την θέση του στοιχείου αν το βρήκε ή -1. Πως;

```
int find(int haystack[100], int needle) {  
    int i;  
    for(i = 0; i < 100; i++) {  
        if (haystack[i] == needle) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Γραμμική / Σειριακή Αναζήτηση (Linear / Serial Search)

Η Γραμμική Αναζήτηση (Linear Search) είναι ένας αλγόριθμος αναζήτησης ενός στοιχείου σε μια ακολουθία στοιχείων.

Ο αλγόριθμος ξεκινά με το πρώτο στοιχείο της ακολουθίας και σε κάθε βήμα του συγκρίνει το στοιχείο μας με αυτό της ακολουθίας. Αν είναι ίσα, σταματάει, αλλιώς συνεχίζει στο επόμενο σειριακά μέχρι να φτάσει στο τελευταίο στοιχείο.

Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(n)$ ενώ χρειάζεται σταθερό χώρο μνήμης $O(1)$

Μπορούμε να βρούμε ένα στοιχείο πιο γρήγορα
από $O(n)$;

Ταξινόμηση

Μια ακολουθία στοιχείων a_i λέγεται **ταξινομημένη** (sorted) ως προς κάποιο τελεστή σύγκρισης \leq_α , αν και μόνο αν $\forall i \leq j. a_i \leq_\alpha a_j$

π.χ., η ακολουθία 1, 7, 8, 10, 19 είναι ταξινομημένη ως προς έναν τελεστή \leq_α εάν είναι ο τελεστή σύγκρισης ακεραίων: $a_i \leq_\alpha a_j \Leftrightarrow a_i \leq a_j$ (αύξουσα)

π.χ., η ακολουθία 19, 10, 8, 7, 1 είναι ταξινομημένη ως προς έναν τελεστή \leq_ϕ που ορίζεται ως εξής: $a_i \leq_\phi a_j \Leftrightarrow -a_i \leq -a_j \Leftrightarrow a_j \leq a_i$ (φθίνουσα)

Δυαδική Αναζήτηση (Binary Search)

Η Δυαδική Αναζήτηση (Binary Search) είναι ένας αλγόριθμος αναζήτησης που μας επιτρέπει να βρούμε αν ένα στοιχείο βρίσκεται σε μια ακολουθία ταξινομημένων στοιχείων.

Σε κάθε βήμα του αλγορίθμου, η ακολουθία χωρίζεται σε **δύο τμήματα** και συγκρίνουμε το στοιχείο μας με το μεσαίο στοιχείο της ακολουθίας. Ο αλγόριθμος συνεχίζει σε ένα από τα δύο τμήματα μέχρι να βρεθεί το στοιχείο ή να δείξουμε ότι το στοιχείο δεν μπορεί να βρίσκεται σε αυτό το τμήμα.

Θέλω μια συνάρτηση που να δέχεται έναν πίνακα ή
ταξινομημένων ακεραίων σε αύξουσα σειρά και έναν ακέραιο
που ψάχνουμε και να γυρνάει αν υπάρχει στον πίνακα ή όχι. Πως;

Δυαδική Αναζήτηση (Binary Search)

```
int binary_search(int elem, int *array, int n) {  
    int mid, low = 0, high = n - 1;  
  
    while (low <= high) {  
        mid = low + (high - low) / 2;  
  
        if (array[mid] == elem)  
            return 1;  
  
        else if (array[mid] < elem)  
            low = mid + 1;  
  
        else  
            high = mid - 1;  
    }  
  
    return 0;  
}
```

Τι πολυπλοκότητα έχει αυτός ο αλγόριθμος;

Χρόνος: $O(\log n)$
Χώρος: $O(1)$

Υλοποιημένη στην συνάρτηση
bsearch της stdlib.h

Διάλεξη 18 - Ταξινόμηση και Δεδομένα Εισόδου #2 (Αρχεία)

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Το Instagram έχει 2 δισεκατομμύρια χρήστες σε έναν πίνακα ακεραίων (έστω ένας ακέραιος ανά χρήστη). Πόσα βήματα (χρονική πολυπλοκότητα) χρειάζεστε για να βρείτε αν ο χρήστης 424242 βρίσκεται στον πίνακα;

Αλγόριθμοι Ταξινόμησης (Sorting Algorithms)

1. Bubblesort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quicksort

Γράψτε μια συνάρτηση swap που ανταλλάζει δύο ακεραίους

```
#include <stdio.h>

int main() {
    int a = 100, b = 200;
    printf("%d %d\n", a, b);
    swap( ... );
    printf("%d %d\n", a, b);
    return 0;
}
```

Γράψτε μια συνάρτηση swap που ανταλλάζει δύο ακεραίους

```
#include <stdio.h>

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int a = 100, b = 200;
    printf("%d %d\n", a, b);
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

Ταξινόμηση Επιλογής (Selection Sort)

```
void selection_sort(int n, int *x) {  
    int i, j, min;  
    for (i = 1 ; i <= n - 1 ; i++) {  
        min = i - 1;  
        for (j = i ; j <= n - 1 ; j++)  
            if (x[j] < x[min])  
                min = j;  
        swap(&x[i-1], &x[min]);  
    }  
}
```

Χρόνος: $O(n^2)$
Χώρος: $O(1)$

Ταξινόμηση Εισαγωγής (Insertion Sort)

```
void insertion_sort(int n, int *x) {  
    int i, j;  
    for (i = 1 ; i <= n - 1 ; i++) {  
        j = i - 1;  
        while (j >= 0 && x[j] > x[j+1]) {  
            swap(&x[j], &x[j+1]);  
            j--;  
        }  
    }  
}
```

Χρόνος: $O(n^2)$
Χώρος: $O(1)$

Ταξινόμηση Φυσαλίδας (Bubblesort)

```
void bubblesort(int n, int *x) {  
    int i, j;  
    for (i = 1 ; i <= n - 1 ; i++)  
        for (j = n - 1 ; j >= i ; j--)  
            if (x[j-1] > x[j])  
                swap(&x[j-1], &x[j]);  
}
```

Χρόνος: $O(n^2)$
Χώρος: $O(1)$

Ταξινόμηση Συγχώνευσης (Merge Sort)

Η ταξινόμηση συγχώνευσης (merge sort) είναι ένας αλγόριθμος divide and conquer (διαιρεί και βασίλευε) που έχει θεωρητικά την καλύτερη πολυπλοκότητα. Ο αλγόριθμος έχει δύο βήματα:

1. Χώρισε τον πίνακα σε δύο υποπίνακες
 - a. κάλεσε ταξινόμηση συγχώνευσης στους υποπίνακες
2. Συγχώνευσε τα στοιχεία των δύο ταξινομημένων υποπινάκων

Ταξινόμηση Συγχώνευσης (Merge Sort)

```
void merge_sort(int *array, int left, int right) {  
    if (left < right) {  
        int middle = left + (right - left) / 2;  
        merge_sort(array, left, middle);  
        merge_sort(array, middle + 1, right);  
        merge(array, left, middle, right);  
    }  
}
```

Ταξινόμηση Συγχώνευσης (Merge Sort)

```
void merge(int *x, int l, int m, int r) {  
    int i, j, k, n1 = m - l + 1, n2 = r - m;  
    int left[n1], right[n2];  
  
    for (i = 0; i < n1; i++) left[i] = x[l + i];  
    for (j = 0; j < n2; j++) right[j] = x[m + 1 + j];  
    i = 0; j = 0; k = l;  
  
    while (i < n1 && j < n2) {  
  
        if (left[i] <= right[j]) x[k++] = left[i++];  
        else x[k++] = right[j++];  
    }  
  
    while (i < n1) x[k++] = left[i++];  
    while (j < n2) x[k++] = right[j++];  
}
```

Χρόνος: $O(n \log n)$
Χώρος: $O(n)$

Thanks John von Neumann

Ταχυταξινόμηση (Quicksort)

Η ταξινόμηση ταχυταξινόμηση (quicksort) είναι ένας αλγόριθμος divide and conquer (διαίρει και βασίλευε) που είναι **ιδιαίτερα δημοφιλής**. Ο αλγόριθμος έχει τρία βήματα:

1. Διάλεξε (έστω τυχαία) το στοιχείο διαμέρισης του πίνακα (pivot element)
2. Διαμέρισε τον πίνακα σε δύο υποπίνακες - αριστερά έχει τα στοιχεία που είναι μικρότερα του pivot και δεξιά τα στοιχεία που είναι μεγαλύτερα
3. Τρέξε ταχυταξινόμηση για τους δύο υποπίνακες

Ταχυταξινόμηση (Quicksort)

```
void quicksort (int *x, int lower, int upper) {  
    if (lower < upper) {  
        int pivot = x[(lower + upper) / 2];  
        int i, j;  
        for (i = lower, j = upper; i <= j;) {  
            while (x[i] < pivot) i++;  
            while (x[j] > pivot) j--;  
            if (i <= j) swap(&x[i++], &x[j--]);  
        }  
        quicksort(x, lower, j);  
        quicksort(x, i, upper);  
    }  
}
```

Χρόνος: $O(n^2)$ (worst case), $O(n \log n)$ (average case)
Χώρος: $O(n)$ (εδώ) - γίνεται και σε [O\(\log n\)](#)

Υλοποιημένη στην συνάρτηση
qsort της stdlib.h

Thanks [Tony Hoare](#)

Δεδομένα Εισόδου #2 (Αρχεία)

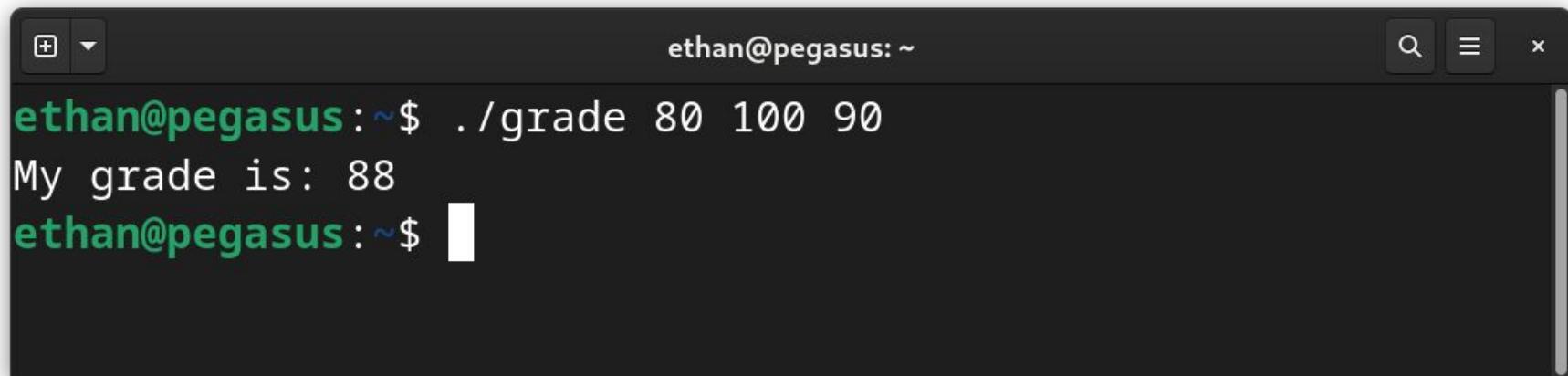
Δεδομένα Εισόδου και Εξόδου (Input and Output Data)



Δεδομένα Εισόδου (Input Data)

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

1. **Ορίσματα** στην γραμμή εντολών



A screenshot of a terminal window titled "ethan@pegasus: ~". The window contains the following text:

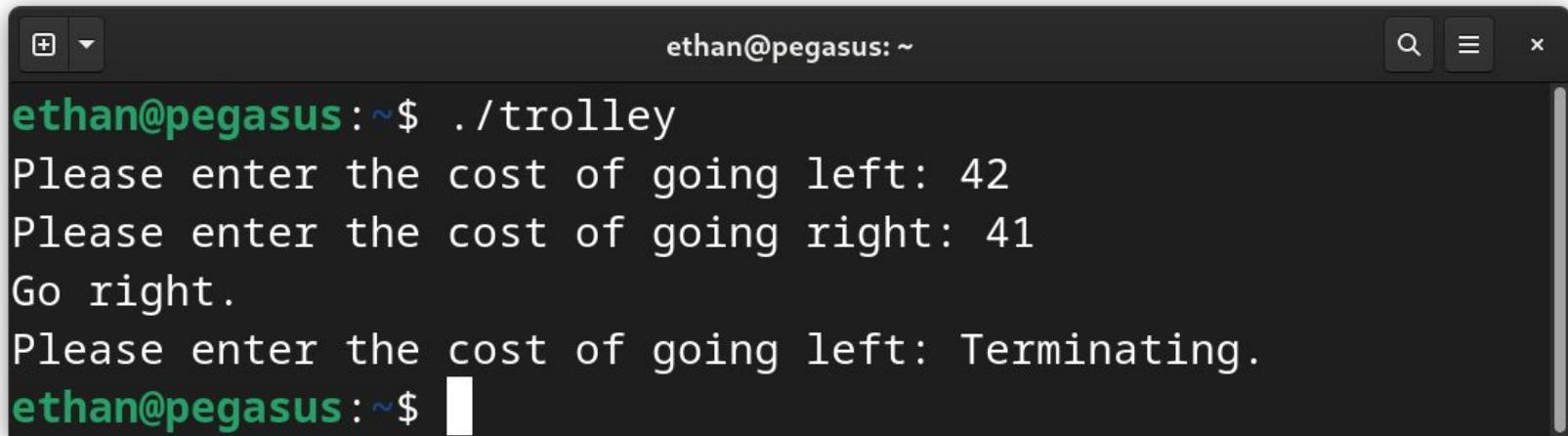
```
ethan@pegasus:~$ ./grade 80 100 90
My grade is: 88
ethan@pegasus:~$
```

The terminal window has a dark theme with light-colored text. The title bar is at the top, and there are standard window controls (minimize, maximize, close) on the right. The main area of the terminal shows the command `./grade` followed by three integers (80, 100, 90), which are then processed by the program to output "My grade is: 88". A cursor is visible at the end of the last line.

Δεδομένα Εισόδου (Input Data) - 2/4

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

2. Γράφοντας κείμενο στην **πρότυπη είσοδο** (standard input ή `stdin`) συνήθως με το πληκτρολόγιο

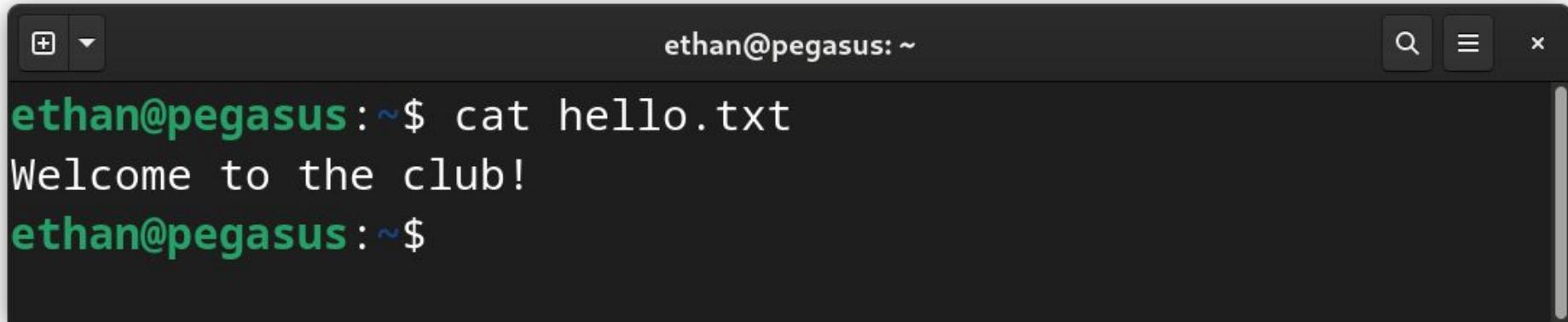


```
ethan@pegasus: ~$ ./trolley
Please enter the cost of going left: 42
Please enter the cost of going right: 41
Go right.
Please enter the cost of going left: Terminating.
ethan@pegasus: ~$
```

Δεδομένα Εισόδου (Input Data) - 3/4

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

3. Διαβάζοντας **αρχεία** από το σύστημα αρχείων (σήμερα!)

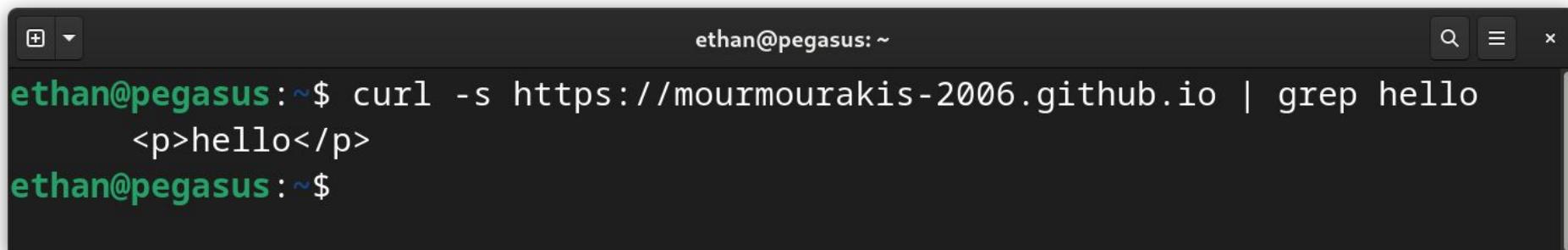


```
ethan@pegasus: ~$ cat hello.txt
Welcome to the club!
ethan@pegasus: ~$
```

Δεδομένα Εισόδου (Input Data) - 4/4

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

4. Διαβάζοντας από το **δίκτυο** ή άλλες πηγές - π.χ., User Interface (σε επόμενα εξάμηνα)



A screenshot of a terminal window titled "ethan@pegasus: ~". The window contains the following command and its output:

```
ethan@pegasus:~$ curl -s https://mourmourakis-2006.github.io | grep hello
<p>hello</p>
ethan@pegasus:~$
```

Δεδομένα Εισόδου (Input Data)

Τα **δεδομένα εισόδου** (input data) είναι μια σειρά από χαρακτήρες (bytes) τα οποία ο χρήστης δίνει στο πρόγραμμα. Υπάρχουν 4 μέθοδοι να εισάγουμε δεδομένα:

1. **Ορίσματα** στην γραμμή εντολών 
2. Γράφοντας κείμενο στην **πρότυπη είσοδο** (standard input ή `stdin`) 
3. Διαβάζοντας **αρχεία** από το σύστημα αρχείων (σήμερα!)  
4. Διαβάζοντας από το **δίκτυο** ή άλλες πηγές (άλλα εξάμηνα)

75%



Χρήση της συνάρτησης `scanf` - Άλλοι τύποι ορισμάτων

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>
#include <math.h>
int main() {
    double d1, d2;
    printf("Gimme two doubles: ");
    scanf("%lf %lf", &d1, &d2);
    printf("Hypotenuse: %.1lf\n", sqrt(d1 * d1 + d2 * d2));
    return 0;
}
```

\$./hypotenuse
Gimme two numbers: 3.0 4.0
Result: 5.0

Χρήση της συνάρτησης scanf - Άλλοι τύποι ορισμάτων

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    char message[7];
    printf("Say something: ");
    scanf("%s", message);
    printf("%s\n", message);
    return 0;
}
```

\$./message
Say something: hello!
hello!



Δεν βάλαμε & πριν την μεταβλητή message. Πως και λειτουργεί;

Χρήση της συνάρτησης `scanf` - Άλλοι τύποι ορισμάτων

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    char message[7];
    printf("Say something: ");
    scanf("%s", message);
    printf("%s\n", message);
    return 0;
}
```

\$./message
Say something: Houston,
we've had a problem here.
Houston,
Segmentation fault



Προσοχή! Δεν υπάρχει κανένας έλεγχος ότι δεν θα διαβαστούν περισσότεροι χαρακτήρες από 6

Χρήση της συνάρτησης `scanf` - Άλλοι τύποι ορισμάτων

Τι κάνει το παρακάτω πρόγραμμα;

```
#include <stdio.h>

int main() {
    char message[7];
    printf("Say something: ");
    scanf("%6s", message);
    printf("%s\n", message);
    return 0;
}
```

\$./message
Say something: Houston,
we've had a problem here.
Houston,
Segmentation fault



Μπορούμε να θέσουμε περιορισμό στους πόσους χαρακτήρες θα διαβαστούν

Η συνάρτηση gets

Η συνάρτηση gets συμπεριφέρεται παρόμοια με την scanf("%s") και γενικά αποφεύγεται για λόγους ασφαλείας.

The screenshot shows a terminal window with the following details:

- Title bar: ethan@pegasus: ~
- Left pane: GETS(3)
- Middle pane: Linux Programmer's Manual
- Right pane: GETS(3)

The content of the middle pane is:

NAME
gets - get a string from standard input (DEPRECATED)

SYNOPSIS
`#include <stdio.h>`
`char *gets(char *s);`

DESCRIPTION
Never use this function.

`gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or `EOF`, which it replaces with a null byte

Θέλω οπωσδήποτε να χρησιμοποιήσω scanf; %ms since C11

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

int main(int argc, char ** argv) {
    char *string; int items_read;
    items_read = scanf("%ms", &string);
    if (items_read != 1) {
        fprintf(stderr, "No matching characters\n");
        return 1;
    }
    printf("read the following string: %s\n", string);
    // don't forget to free!
    free(string);
    return 0;
}
```

Χειρισμός Αρχείων (File Handling)

Filesystem (Σύστημα Αρχείων)

Αρχείο (File) είναι ένας πόρος για να καταγράφουμε δεδομένα σε έναν υπολογιστή. Συνήθως αποθηκεύεται στην δευτερεύουσα μνήμη (π.χ., σκληρός δίσκος).

Στο Linux σχεδόν τα πάντα είναι ένα αρχείο.

Κάθε αρχείο:

Το περιεχόμενο του αρχείου είναι ένας πίνακας από χαρακτήρες char bytes []

- Έχει ένα όνομα (filename/basename)
- Βρίσκεται μέσα σε ένα συγκεκριμένο φάκελο (directory/folder)
- Έχει ένα πλήρες μονοπάτι (filepath) που καθορίζει που βρίσκεται το αρχείο.

Παράδειγμα: το filepath ενός αρχείου είναι `/home/users/thanassis/documents/students.txt`, ο φάκελος μέσα στον οποίο βρίσκεται αυτό το αρχείο είναι ο `/home/users/thanassis/documents` ενώ το όνομα του αρχείου είναι `students.txt`. Το `:txt` στο τέλος του ονόματος λέγεται επέκταση (extension) και συνήθως να περιγράφει τον τύπο του αρχείου.

Ο τύπος FILE

Ο τύπος **FILE** ορίζεται στην κεφαλίδα stdio.h και μας επιτρέπει να αναφερόμαστε σε αρχεία που άνοιξε το πρόγραμμά μας.

```
#include <stdio.h>

int main() {
    printf("Size of FILE type: %zu\n", sizeof(FILE));
    return 0;
}

$ ./filedef
216
```

216 bytes σε ένα σύστημα Debian! Τι περιέχει εντός;
Πολλά και διάφορα ίσως το συζητήσουμε άλλη φορά

Η συνάρτηση `fopen`

Η συνάρτηση **fopen** επιτρέπει στο πρόγραμμά μας να ανοίξει ένα αρχείο και να επιστρέψει έναν δείκτη σε FILE. Επιστρέφει NULL αν αποτύχει να ανοίξει το αρχείο για οποιοδήποτε λόγο. Η δήλωση της συνάρτησης είναι

```
FILE *fopen(const char *restrict pathname, const char *restrict mode);
```

pathname: το μονοπάτι που αντιστοιχεί στο αρχείο

mode: με ποιο τρόπο να ανοίξουμε το αρχείο (διάβασμα ή γράψιμο;)



ethan@pegasus: ~



DESCRIPTION

The `fopen()` function opens the file whose name is the string pointed to by `pathname` and associates a stream with it.

The argument `mode` points to a string beginning with one of the following sequences (possibly followed by additional characters, as described below):

- `r` Open text file for reading. The stream is positioned at the beginning of the file.
- `r+` Open for reading and writing. The stream is positioned at the beginning of the file.
- `w` Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- `w+` Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- `a` Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- `a+` Open for reading and appending (writing at end of file). The file is created if it does not exist. Output is always appended to the end of the file. POSIX is silent on what the initial read position is when using this mode. For glibc, the initial file position for reading is at the beginning of the file, but for Android/BSD/MacOS, the initial file position for reading is at the end of the file.

Παράδειγμα: Άνοιγμα Αρχείων

```
#include <stdio.h>

int main() {

    FILE *fileToRead, *fileToWrite;
    fileToRead = fopen("input.txt", "r");
    if (!fileToRead) {

        return 1;
    }

    fileToWrite = fopen("output.txt", "w");

    if (!fileToWrite) {

        return 1;
    }

    return 0;
}
```

Ελέγχουμε πάντα το αποτέλεσμα της fopen αν είναι NULL. Για ποιο λόγο μπορεί να αποτύχει;

Παράδειγμα: Άνοιγμα Αρχείων

```
#include <stdio.h>

int main() {

    FILE *fileToRead, *fileToWrite;
    fileToRead = fopen("input.txt", "r");
    if (!fileToRead) {

        return 1;
    }

    fileToWrite = fopen("output.txt", "w");
    if (!fileToWrite) {

        return 1;
    }

    return 0;
}
```

Ελέγχουμε πάντα το αποτέλεσμα της fopen αν είναι NULL. Για ποιο λόγο μπορεί να αποτύχει;

1. Το αρχείο δεν υπάρχει
2. Ο φάκελος δεν υπάρχει
3. Δεν έχουμε ελεύθερο δίσκο για να γράψουμε αρχείο!
4. Δεν έχουμε δικαιώματα να φτιάξουμε αρχείο
5. Έχουμε ανοίξει τον μέγιστο επιτρεπτό αριθμό αρχείων
6. ...

Η συνάρτηση `fclose`

Η συνάρτηση `fclose` μας επιτρέπει να κλείσουμε ένα αρχείο. Επιστρέφει 0 αν επιτύχει ή EOF αν αποτύχει. Η δήλωση της συνάρτησης είναι

```
int fclose(FILE *stream);
```

Πάντα κλείνουμε τα αρχεία που άνοιξε το πρόγραμμά μας αφού τελειώσουμε με την χρήση τους.

Παράδειγμα: Κλείσιμο Αρχείων

```
#include <stdio.h>

int main() {

    FILE *fileToRead, *fileToWrite;

    fileToRead = fopen("input.txt", "r");

    ...

    fileToWrite = fopen("output.txt", "w");

    ...

    fclose(fileToRead);

    fclose(fileToWrite);

    return 0;
}
```

Σε κάθε fopen πρέπει να αντιστοιχεί ένα fclose

Η συνάρτηση **fread**

Η συνάρτηση **fread** μας επιτρέπει να διαβάσουμε bytes από ένα ανοιχτό αρχείο. Επιστρέφει το πλήθος των δεδομένων που διαβάστηκαν. Η δήλωση της συνάρτησης είναι

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *restrict stream);
```

ptr: δείκτης στην μνήμη όπου θα αποθηκευτούν τα δεδομένα εισόδου

size: ο αριθμός bytes κάθε δεδομένου που θα διαβαστεί

nmemb: πόσα δεδομένα να προσπαθήσει να διαβάσει

stream: δείκτης στο ανοιχτό αρχείο

Παράδειγμα: Διάβασμα Αρχείων

```
#include <stdio.h>

int main() {
    FILE *fileToRead, *fileToWrite;
    fileToRead = fopen("input.txt", "r");
    if (!fileToRead) return 1;

    char buffer[1024];

    size_t bytesRead = fread(buffer, sizeof(char), 1023, fileToRead);
    buffer[bytesRead] = '\0';

    printf("# of bytes read: %d\n", bytesRead);
    printf("String read: %s\n", buffer);

    fclose(fileToRead);

    return 0;
}
```

```
$ ./fread
$ echo hello > input.txt
$ ./fread
# of bytes read: 6
String read: hello
```

Παράδειγμα #2: Διάβασμα Αρχείων

```
#include <stdio.h>

int main() {
    FILE *fileToRead, *fileToWrite;
    fileToRead = fopen("input.txt", "r");
    if (!fileToRead) return 1;
    int buffer[1024];
    size_t integersRead = fread(buffer, sizeof(int), 1023, fileToRead);
    printf("# of integers read: %d\n", integersRead);
    printf("Integer read: %d %08x\n", buffer[0], buffer[0]);
    fclose(fileToRead);
    return 0;
}
```

Παράδειγμα #2: Διάβασμα Αρχείων

```
#include <stdio.h>

int main() {
    FILE *fileToRead, *fileToWrite;
    fileToRead = fopen("input.txt", "r");

    if (!fileToRead) return 1;
    int buffer[1024];

    size_t integersRead = fread(buffer, sizeof(int), 1023, fileToRead);
    printf("# of integers read: %d\n", integersRead);
    printf("Integer read: %d %08x\n", buffer[0], buffer[0]);
    fclose(fileToRead);
    return 0;
}
```

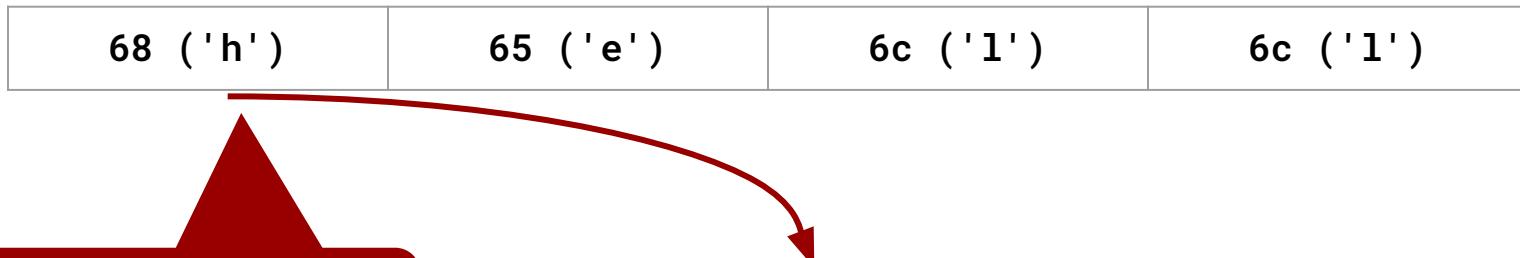
```
$ ./freadint
# of integers read: 1
Integer read: 1819043176 6c6c6568
```

What?!

```
$ hexdump -C input.txt
00000000  68 65 6c 6c 6f 0a  |hello.|
```

Endianness

Endianness λέμε τον τρόπο με τον οποίο οι ακέραιοι αποθηκεύονται στην μνήμη. Οι ακέραιοι αποτελούνται από πολλά bytes και επομένως πρέπει να αποφασίσουμε αν τους αποθηκεύουμε από το μικρότερο στο μεγαλύτερο (little endian) ή από το μεγαλύτερο στο μικρότερο (big endian).



Το πρώτο byte στην μνήμη είναι
το μικρότερο byte του αριθμού

Παράδειγμα Endianness

Τι θα τυπώσει το παρακάτω:

```
#include <stdio.h>

int main() {
    int x = 42;
    char * bytes = (char*)&x;
    int i;
    for(i = 0; i < sizeof(int) / sizeof(char); i++)
        printf("%02x\n", bytes[i]);
    return 0;
}
```

```
$ ./int
2a
00
00
00
```

Η συνάρτηση **fwrite**

Η συνάρτηση **fwrite** μας επιτρέπει να γράψουμε έναν αριθμό δεδομένων σε ένα αρχείο. Επιστρέφει τον αριθμό των στοιχείων που γράφτηκαν. Η δήλωση της συνάρτησης είναι

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *restrict stream);
```

ptr: δείκτης στην μνήμη απ'όπου θα διαβάσουμε τα δεδομένα εξόδου

size: ο αριθμός bytes κάθε δεδομένου που θα γραφτεί

nmemb: πόσα δεδομένα να προσπαθήσει να γράψει

stream: δείκτης στο ανοιχτό αρχείο

Παράδειγμα: Γράψιμο Αρχείων

```
#include <stdio.h>

int main() {
    FILE *fileToWrite;
    fileToWrite = fopen("output.txt", "w");
    if (!fileToWrite) return 1;
    int numbers[4] = {0x42, 0x43, 0x44, 0x45};
    size_t numsWritten = fwrite(numbers, sizeof(int), 4, fileToWrite);
    printf("Wrote: %zu numbers\n", numsWritten);
    fclose(fileToWrite);
    return 0;
}
```

\$./fwrite
Wrote: 4 numbers
\$ hexdump -C output.txt
00000000 42 00 00 00 43 00 00 00 44 00 00 00 45 00 00 00
|B...C...D...E...|

File Descriptor (FD)

Κάθε ανοιχτό αρχείο για ένα πρόγραμμα έχει έναν μοναδικό αριθμό που λέγεται file descriptor. Μπορούμε να βρούμε αυτόν τον αριθμό με την συνάρτηση `fileno`.

```
FILE *fileToRead, *fileToWrite;  
  
fileToRead = fopen("input.txt", "r");  
fileToWrite = fopen("output.txt", "w");  
// ...  
  
printf("fileToRead: %d\n", fileno(fileToRead));  
printf("fileToWrite: %d\n", fileno(fileToWrite));  
printf("stdin: %d\n", fileno(stdin));  
printf("stdout: %d\n", fileno(stdout));  
printf("stderr: %d\n", fileno(stderr));
```

```
$ ./fileno  
fileToRead: 3  
fileToWrite: 4  
stdin: 0  
stdout: 1  
stderr: 2
```

stdin, stdout και stderr

Τα **stdin**, **stdout** και **stderr** είναι δείκτες σε ανοικτά αρχεία που αρχικοποιούνται όταν το πρόγραμμα ξεκινά την εκτέλεσή του και κλείνουν όταν τερματίζει.

stdin: αντιστοιχεί στην πρότυπη είσοδο του προγράμματος και συνήθως έχει file descriptor 0.

stdout: αντιστοιχεί στην πρότυπη έξοδο του προγράμματος και συνήθως έχει file descriptor 1.

stderr: αντιστοιχεί στην έξοδο σφάλματος του προγράμματος και συνήθως έχει file descriptor 2.

Σύγκρινε το `find / -name foo` με το `find / -name foo 2> error.txt`

Η συνάρτηση **fprintf**

Η συνάρτηση **fprintf** είναι όμοια με την printf, απλά αντί να γράφει στο stdout, μπορεί να γράψει σε οποιοδήποτε αρχείο. Η συνάρτηση έχει την ακόλουθη μορφή:

```
int fprintf(FILE *stream, const char *format, ...);
```

Η κλήση printf(x, y, z) είναι ουσιαστικά ισοδύναμη με την fprintf(stdout, x, y, z)

Παράδειγμα χρήσης fprintf/fscanf

```
#include <stdio.h>

int main() {
    FILE *fileToRead, *fileToWrite;
    fileToRead = fopen("input.txt", "r");
    fileToWrite = fopen("output.txt", "w");
    // ...
    int num;
    fscanf(fileToRead, "%d", &num);
    fprintf(fileToWrite, "Number: %d\n", num);
    // ...
    return 0;
}
```

```
$ echo "      42" > input.txt
$ ./stream
$ cat output.txt
Number: 42
```

Άλλες χρήσιμες συναρτήσεις

```
char *fgets(char *s, int size, FILE *stream);
```

```
int feof(FILE *stream);
```

```
int fseek(FILE *stream, long offset, int whence);
```

```
int fgetc(FILE *stream);
```

```
int fputc(int c, FILE *stream);
```

Διάλεξη 19 - Δομές

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Εισαγωγή στις Δομές

Μέχρι στιγμής είδαμε βασικούς τύπους

`int`, `double`, `char`, δείκτες, πίνακες

όμως τα δεδομένα μπορεί να είναι πιο
δομημένα/περίπλοκα από έναν πίνακα ακεραίων. Τι
κάνουμε για αυτά;

Δομή (Struct)

Δομή (struct) (ή αλλιώς εγγραφή / record) στην C λέγεται μια συλλογή πεδίων που συνήθως χρησιμοποιούνται για την ομαδοποίηση πληροφορίας που περιγράφει μια λογική οντότητα.

Για παράδειγμα, έστω ότι θες να γράψεις ένα πρόγραμμα διαχείρισης μιας λίστας φοιτητών. Θα χρειαστείς μεταβλητές για τα ακόλουθα:

```
char first_name[128];  
char last_name[128];  
unsigned int year;  
double grade;
```

Έστω ότι έχουμε 100 φοιτητές, τι θα κάνουμε για να τους αναπαραστήσουμε στο πρόγραμμά μας;

Δήλωση Δομής (Struct Declaration)

Η δήλωση δομής επιτρέπει να δημιουργούμε τους **δικούς μας** (user-defined) τύπους μεταβλητών που μπορούν να έχουν μια συλλογή από τα πεδία που επιθυμούμε.

struct όνομα {

τύπος1 πεδίο1;

τύπος2 πεδίο2;

τύπος3 πεδίο3;

...

} ;

To keyword
struct
υποδηλώνει
ότι ορίζουμε
μια δομή

Το όνομα (ή αλλιώς ετικέτα / tag) της δομής μας επιτρέπει να αναφερόμαστε σε αυτήν



Τα πεδία / fields της δομής περιέχουν ορισμούς τύπων που αποτελούν την δομή

Δήλωση Τύπου Δομής (Struct Type Declaration)

Η δήλωση δομής επιτρέπει να δημιουργούμε τους **δικούς μας** (user-defined) τύπους μεταβλητών που μπορούν να έχουν μια συλλογή από τα πεδία που επιθυμούμε.

```
struct student {  
    char first_name[128];  
    char last_name[128];  
    unsigned int year;  
    double grade;  
};
```

To keyword
struct
υποδηλώνει
ότι ορίζουμε
μια δομή

Το όνομα (ή αλλιώς ετικέτα / tag) της δομής μας επιτρέπει να αναφερόμαστε σε αυτήν

Τα πεδία / fields της δομής περιέχουν ορισμούς τύπων που αποτελούν την δομή

Δήλωση Μεταβλητής Τύπου Δομής

Δηλώνουμε μια μεταβλητή με τύπο δομής ως εξής:

```
struct όνομα_δομής όνομα_μεταβλητής;
```

Για παράδειγμα:

```
struct student st1, st2;
```

Δύο μεταβλητές st1, st2
ΤΥΠΟΥ struct student

```
struct student *student_ptr;
```

Δείκτης σε δομή τύπου struct
student

```
struct student student_array[100];
```

Πίνακας με 100 δομές τύπου
struct student

Προσπέλαση Πεδίων Δομής (Struct Field Access)

Για να προσπελάσουμε το πεδίο μιας δομής χρησιμοποιούμε την σύνταξη:

όνομα_μεταβλητής_τύπου_δομής.όνομα_πεδίου

Για παράδειγμα:

st1.year

Αναφέρεται στο ακέραιο
πεδίο **year** της δομής
student που αντιστοιχεί στην
μεταβλητή st1

Το πεδίο μπορεί να χρησιμοποιηθεί όπως μια μεταβλητή στην C

Ανάθεση και Χρήση Πεδίων (Field Assignment and Usage)

```
int main() {  
    struct student st1;  
  
    st1.year = 1;  
    st1.grade = 7.54;  
  
    strncpy(st1.first_name, "Thanos", sizeof(st1.first_name) - 1);  
    st1.first_name[sizeof(st1.first_name) - 1] = '\0';  
  
    strncpy(st1.last_name, "Barbounis", sizeof(st1.last_name) - 1);  
    st1.last_name[sizeof(st1.last_name) - 1] = '\0';  
  
    printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade, st1.year);  
  
    return 0;  
}
```

Τι τυπώνει αυτό το πρόγραμμα;

Ανάθεση και Χρήση Πεδίων (Field Assignment and Usage)

```
int main() {  
    struct student st1;  
  
    st1.year = 1;  
    st1.grade = 7.54;  
  
    strncpy(st1.first_name, "Thanos", sizeof(st1.first_name) - 1);  
    st1.first_name[sizeof(st1.first_name) - 1] = '\0';  
  
    strncpy(st1.last_name, "Barbounis", sizeof(st1.last_name) - 1);  
    st1.last_name[sizeof(st1.last_name) - 1] = '\0';  
  
    printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade, st1.year);  
  
    return 0;  
}
```

Τι τυπώνει αυτό το πρόγραμμα;

Ανάθεση τιμών στα πεδία
της δομής (integer, double,
char[128])

Τύπωμα των τιμών των
πεδίων της δομής

\$./struct
Thanos Barbounis: 7.540000 [1 year]

Αρχικοποίηση Δομής (Struct Initialization)

Χρησιμοποιώντας σύνταξη παρόμοια με την αρχικοποίηση πινάκων μπορούμε να αρχικοποιήσουμε δομές:

```
struct student st1;

struct student st1 = {
    "Thanos",
    "Barbounis",
    1,
    7.54
};

printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade,
st1.year);
```

Αρχικοποίηση Δομής (Struct Initialization)

Χρησιμοποιώντας σύνταξη παρόμοια με την αρχικοποίηση πινάκων μπορούμε να αρχικοποιήσουμε δομές:

```
struct student st1;

struct student st1 = {    Ιδια σειρά με την
    "Thanos",           δήλωση της δομής      struct student {
    "Barbounis",         ←→                         char first_name[128];
    1,                  ←→                         char last_name[128];
    7.54                ←→                         unsigned int year;
};

printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade,
st1.year);
```

```
$ ./struct
Thanos Barbounis: 7.540000 [1 year]
```

Αρχικοποίηση Δομής (Struct Initialization)

Χρησιμοποιώντας σύνταξη παρόμοια με την αρχικοποίηση πινάκων μπορούμε να αρχικοποιήσουμε δομές:

```
struct student st1;

struct student st1 = {
    "Thanos",
    "Barbounis",
    1
};

printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade,
st1.year);

$ ./struct
Thanos Barbounis: 0.000000 [1 year]
```

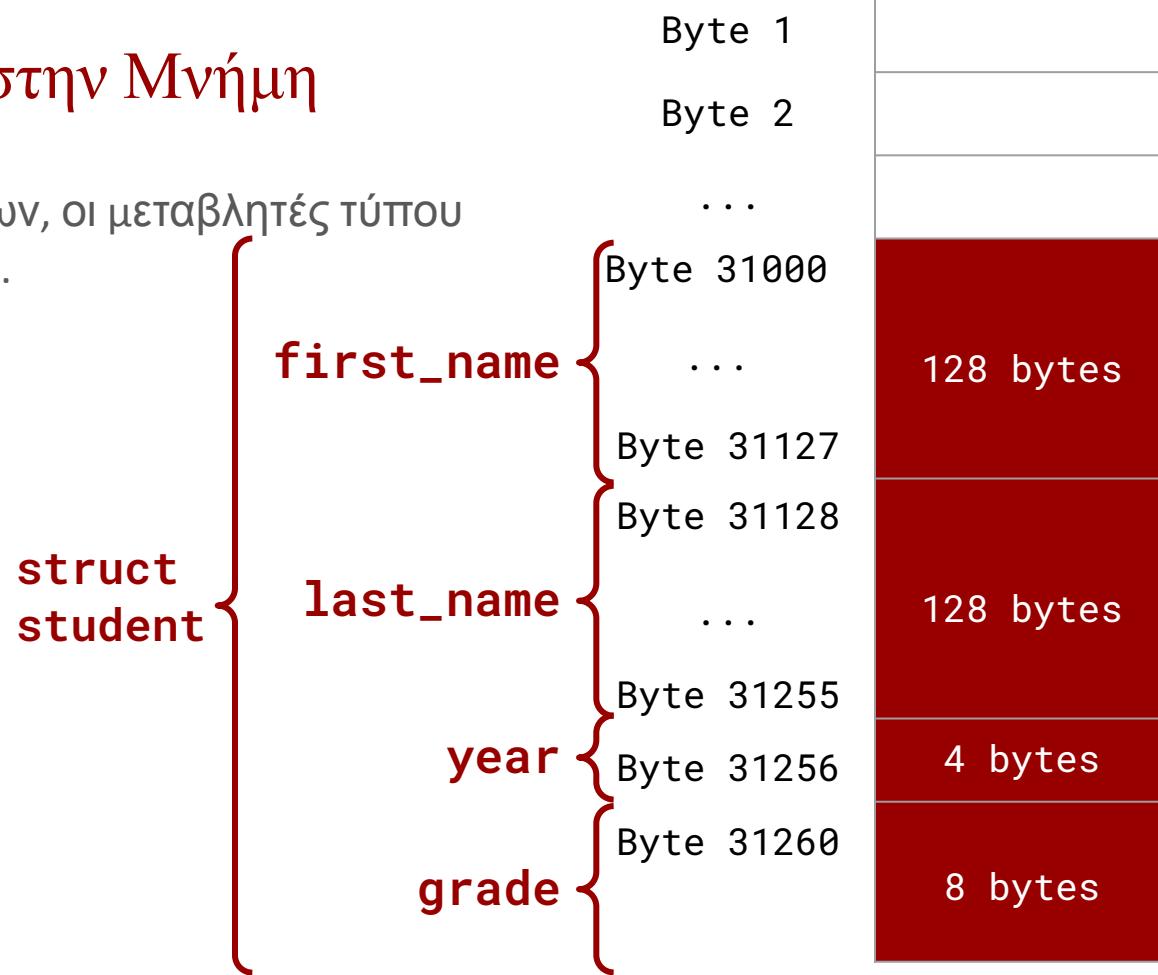


Αν παραλείψουμε μια τιμή αρχικοποιείται στο 0

Αναπαράσταση Δομής στην Μνήμη

Όπως οι μεταβλητές άλλων τύπων, οι μεταβλητές τύπου δομής πιάνουν bytes στην μνήμη.

```
struct student {  
    char first_name[128];  
    char last_name[128];  
    unsigned int year;  
    double grade;  
} st1;
```

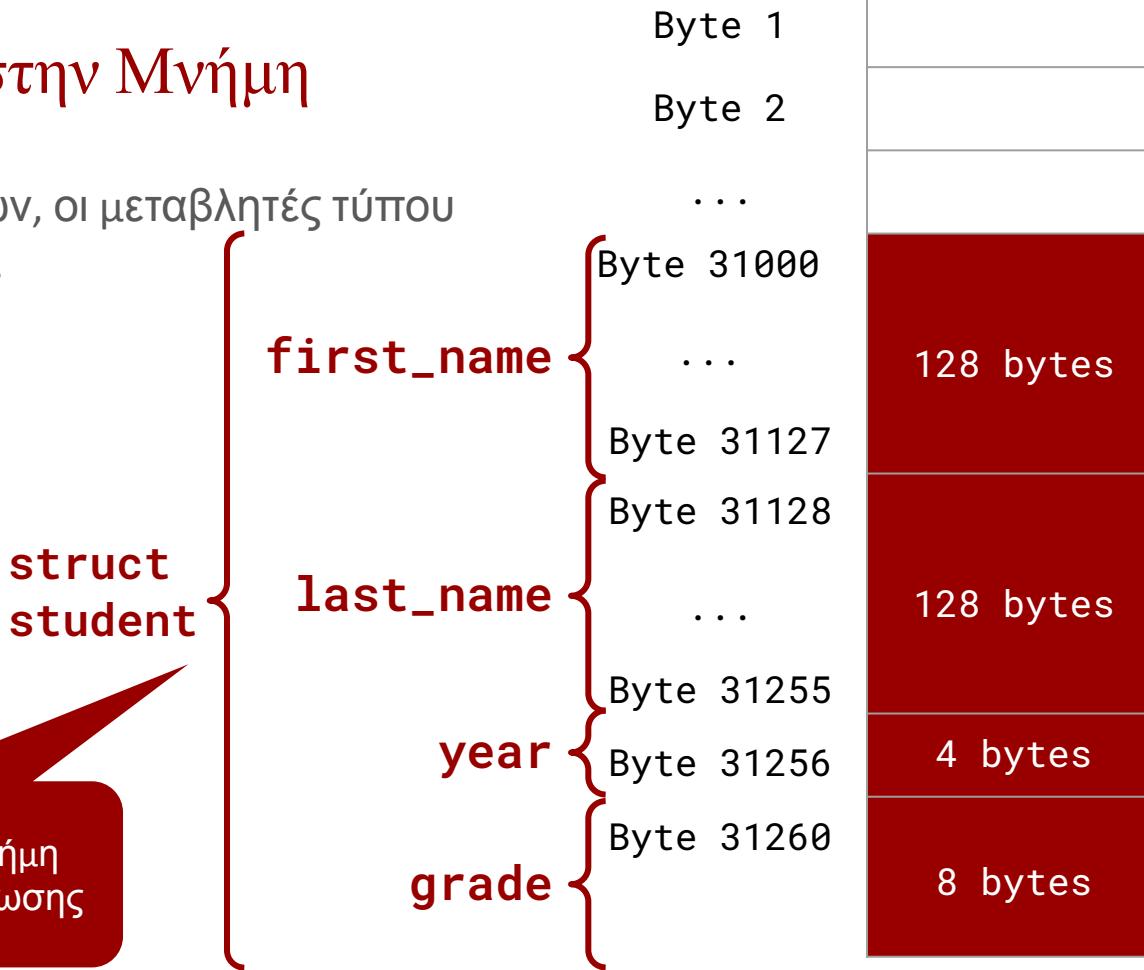


Αναπαράσταση Δομής στην Μνήμη

Όπως οι μεταβλητές άλλων τύπων, οι μεταβλητές τύπου δομής πιάνουν bytes στην μνήμη.

```
struct student {  
    char first_name[128];  
    char last_name[128];  
    unsigned int year;  
    double grade;  
} st1;
```

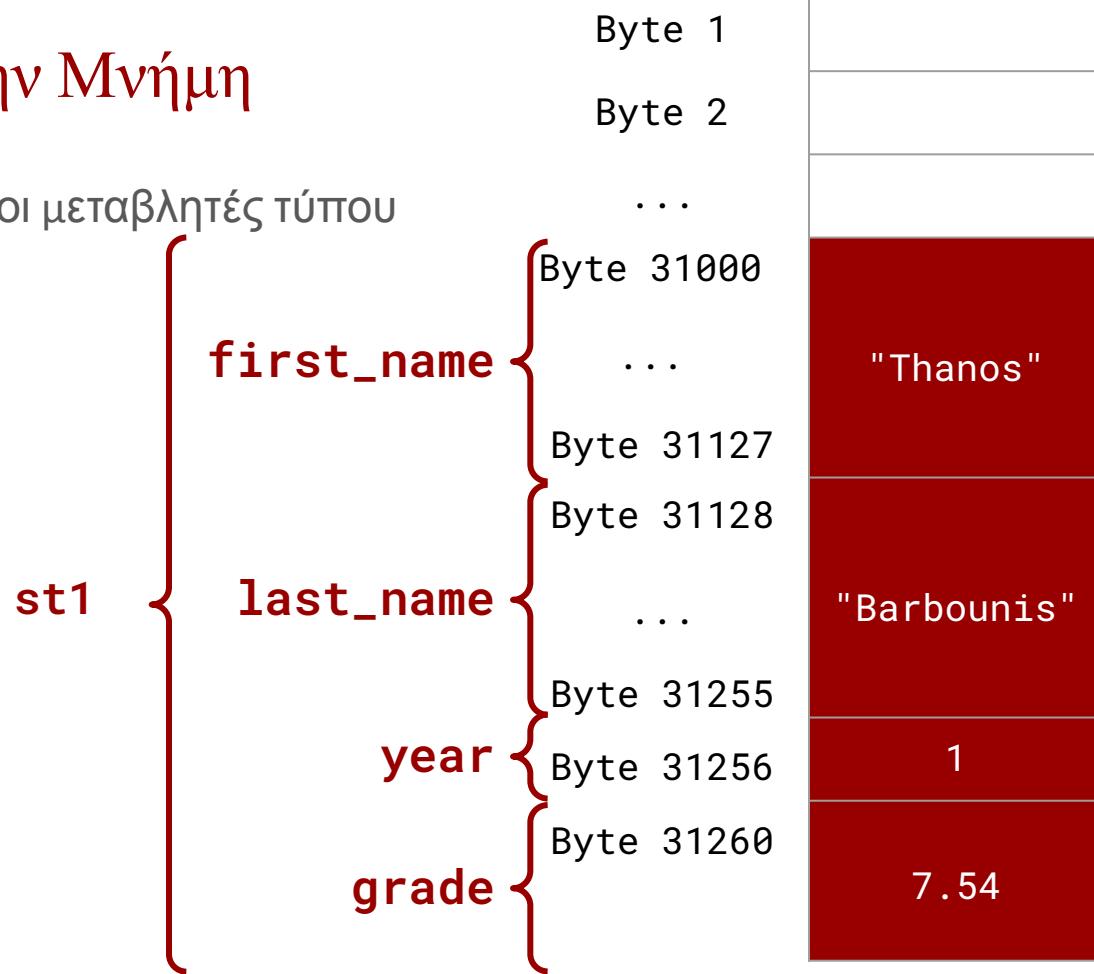
Σειρά πεδίων στην μνήμη
όμοια με αυτήν της δήλωσης



Αναπαράσταση Δομής στην Μνήμη

Όπως οι μεταβλητές άλλων τύπων, οι μεταβλητές τύπου δομής πιάνουν bytes στην μνήμη.

```
struct student st1 = {  
    "Thanos",  
    "Barbounis",  
    1,  
    7.54  
};
```

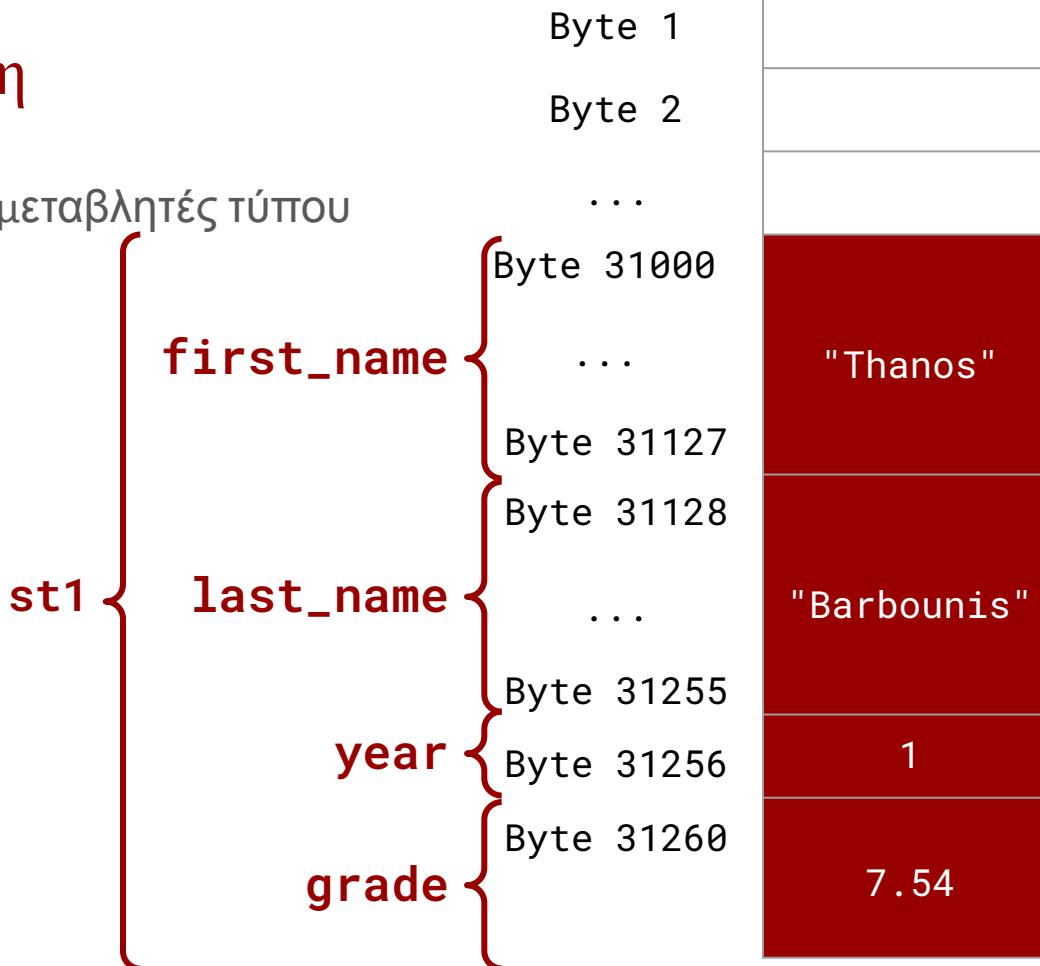


Μέγεθος Δομής στην Μνήμη

Όπως οι μεταβλητές άλλων τύπων, οι μεταβλητές τύπου δομής πιάνουν bytes στην μνήμη.

Τι θα τυπώσει το ακόλουθο;

```
printf("%zu\n",
      sizeof(struct student));
```



Μέγεθος Δομής στην Μνήμη

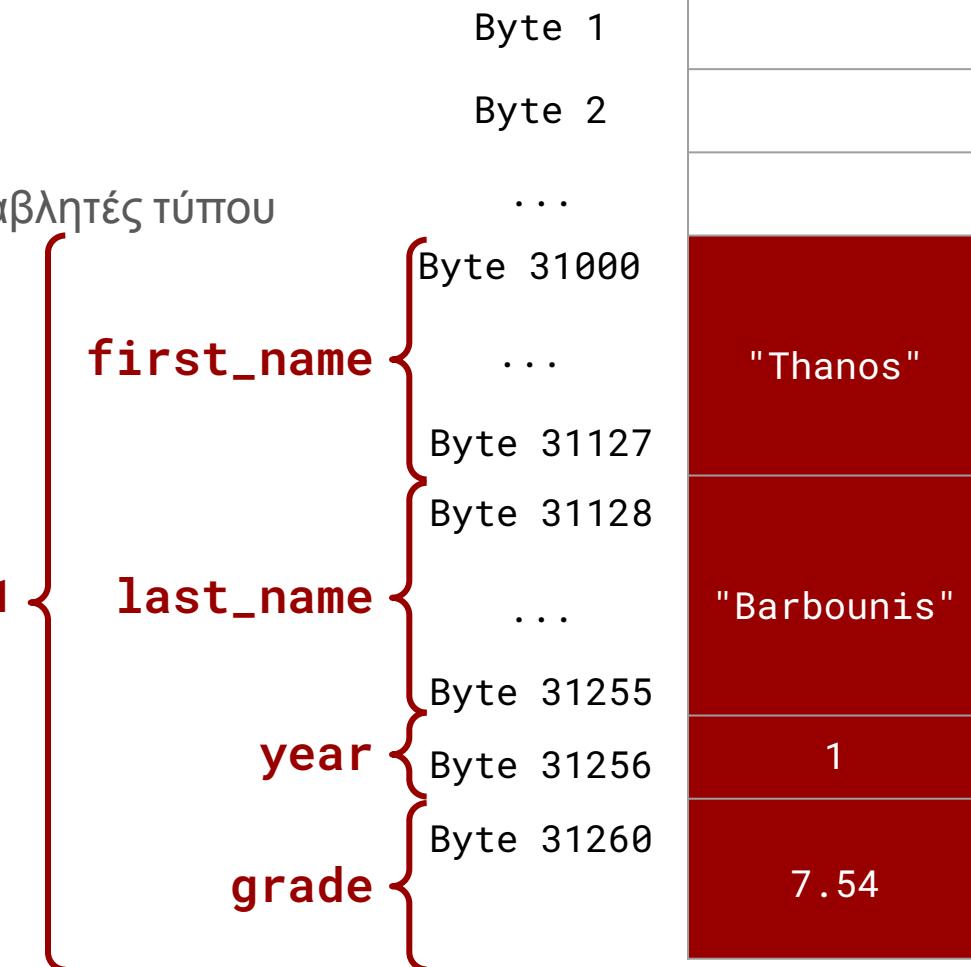
Όπως οι μεταβλητές άλλων τύπων, οι μεταβλητές τύπου δομής πιάνουν bytes στην μνήμη.

Τι θα τυπώσει το ακόλουθο;

```
printf("%zu\n",
```

```
      sizeof(struct student));
```

```
$ gcc -m32 -o struct struct.c
$ ./struct
268
```



Μέγεθος Δομής στην Μνήμη - Padding

```
#include <stdio.h>
int main() {
    struct pixel_tag {
        char red;
        char green;
        char blue;
        int alpha;
    } pixel = {0xFF, 0xFF, 0xFF, 42};
    printf("%zu\n", sizeof(pixel));
    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

\$ gcc -m32 -o struct2 struct2.c
\$./struct2
8
???

Μέγεθος Δομής στην Μνήμη - Padding

Μόνο 7 από τα 8 bytes
χρησιμοποιούνται

```
#include <stdio.h>
int main() {
    struct pixel_tag {
        char red;
        char green;
        char blue;
        int alpha;
    } pixel = {0xFF, 0xFF, 0xFF, 42};
    printf("%zu\n", sizeof(pixel));
    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

```
$ gcc -m32 -o struct2 struct2.c
$ ./struct2
8
```

red	0xFF
green	0xFF
blue	0xFF
padding	(κενό)
alpha	42

Ο μεταγλωτιστής μπορεί να αποφασίσει να προσθέσει padding ("κενά") που δεν χρησιμοποιείται προκειμένου οι διευθύνσεις των πεδίων να είναι πολλαπλάσιο του 4 (ή του 8 / sizeof(void*)) για λόγους απόδοσης ([memory alignment](#))

Μέγεθος Δομής στην Μνήμη - Padding

```
#include <stdio.h>
int main() {
    struct pixel_tag {
        char red; int alpha;
        char green; int beta;
        char blue; int gamma;
    } pixel;
    printf("%zu\n", sizeof(pixel));
    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

Μέγεθος Δομής στην Μνήμη - Padding

```
#include <stdio.h>
int main() {
    struct pixel_tag {
        char red; int alpha;
        char green; int beta;
        char blue; int gamma;
    } pixel;
    printf("%zu\n", sizeof(pixel));
    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

```
$ gcc -m32 -o struct3 struct3.c
$ ./struct3
24
```

red

padding
(κενό)

alpha

green

padding
(κενό)

Μέγεθος Δομής στην Μνήμη - Padding

```
#include <stdio.h>
int main() {
    struct pixel_tag {
        char red; int alpha;
        char green; int beta;
        char blue; int gamma;
    } pixel;
    printf("%zu\n", sizeof(pixel));
    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

```
$ gcc -m32 -o struct3 struct3.c
$ ./struct3
24
```

Μόνο 15 από τα 24 bytes
χρησιμοποιούνται

...

Συνοψίζοντας

Ποιο είναι το μέγεθος του **struct student**;

Ότι επιστρέψει το `sizeof(struct student)`

Ανάθεση με Δομές

Για να αντιγραφούν τα περιεχόμενα μιας δομής σε μια άλλη, χρησιμοποιούμε τον τελεστή ανάθεσης:

```
#include <stdio.h>

struct point { int x; int y; };

int main() {
    struct point pt1 = { 3, 4 };

    struct point pt2;

    printf("%d %d\n", pt2.x, pt2.y);

    pt2 = pt1;

    printf("%d %d\n", pt2.x, pt2.y);

    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

Ανάθεση με Δομές

Για να αντιγραφούν τα περιεχόμενα μιας δομής σε μια άλλη, χρησιμοποιούμε τον τελεστή ανάθεσης:

```
#include <stdio.h>

struct point { int x; int y; };

int main() {

    struct point pt1 = { 3, 4 };

    struct point pt2;

    printf("%d %d\n", pt2.x, pt2.y);

    pt2 = pt1;

    printf("%d %d\n", pt2.x, pt2.y);

    return 0;
}
```

Τι θα τυπώσει αυτό το πρόγραμμα;

```
$ ./copy
-29387249 0
3 4
```

Δεν έχει αρχικοποιηθεί οπότε θα τυπώσει
ότι υπήρχε στην μνήμη

Όλα τα περιεχόμενα του pt1
αντιγράφηκαν στο pt2 με την ανάθεση

Ανάθεση με Δομές

Για να αντιγραφούν τα περιεχόμενα μιας δομής σε μια άλλη, πρέπει να ακριβώσ το **ΐδιο όνομα** τύπου:

```
#include <stdio.h>

struct point1 { int x; int y; };

struct point2 { int x; int y; };

int main() {
    struct point1 pt1 = { 3, 4 };
    struct point2 pt2;

    pt2 = pt1;

    printf("%d %d\n", pt2.x, pt2.y);

    return 0;
}
```

```
$ gcc -o badcopy badcopy.c
badcopy.c: In function 'main':
badcopy.c:7:9: error: incompatible types when assigning
to type 'struct point2' from type 'struct point1'
      7 |     pt2 = pt1;
           |           ^~~
```

Σύγκριση με δομές

Μπορώ να συγκρίνω δομές με τελεστές σύγκρισης;

```
struct point pt1 = { 3, 4 };

struct point pt2;

pt2 = pt1;

if (pt1 == pt2) printf("impossible\n");

$ gcc -o copy copy.c
copy.c: In function 'main':
copy.c:13:11: error: invalid operands to binary == (have 'struct point' and
'struct point')
13 |     if (pt1 == pt2) printf("impossible\n");
   |     ^~
```

ΟΧΙ, πρέπει να συγκρίνω τα πεδία της δομής ένα-ένα

Το προσδιοριστικό **typedef**

Το προσδιοριστικό **typedef** (type definition) χρησιμοποιείται για τον ορισμό συνωνύμων για τύπους. Χρησιμοποιούμε την σύνταξη:

```
typedef υπάρχον_τύπος νέος_τύπος;
```

Μετά από αυτόν τον ορισμό οι δύο τύποι είναι συνώνυμοι. Για παράδειγμα:

```
typedef unsigned int myuint;
```

Με αυτόν τον ορισμό οι ακόλουθες δηλώσεις μεταβλητών είναι ισοδύναμες:

```
unsigned int x;  myuint x;
```

Πίνακες και `typedef`

Έστω ότι θέλουμε να ορίσουμε έναν δικό μας τύπο `page` που αντιστοιχεί σε έναν πίνακα 1024 ακεραίων, μπορούμε να ορίσουμε:

```
typedef int page[1024];
```

Με αυτόν τον ορισμό μπορούμε να δηλώσουμε έναν πίνακα ως εξής:

```
page mypage;
```

```
...
printf("%zu\n", sizeof(mypage));
...
$ ./page
4096
```

Structs και typedef

Μπορούμε να δημιουργήσουμε μια συντομογραφία Student για τον τύπο struct student ως εξής:

```
#include <stdio.h>

typedef struct student {
    char first_name[128]; char last_name[128];
    int year; double grade;
} Student;

int main() {
    Student st1 = {"Thanos", "Barbounis", 1, 7.54};
    printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade, st1.year);
    return 0;
}
```



Ορισμός τύπου Student που αντιστοιχεί στο struct student

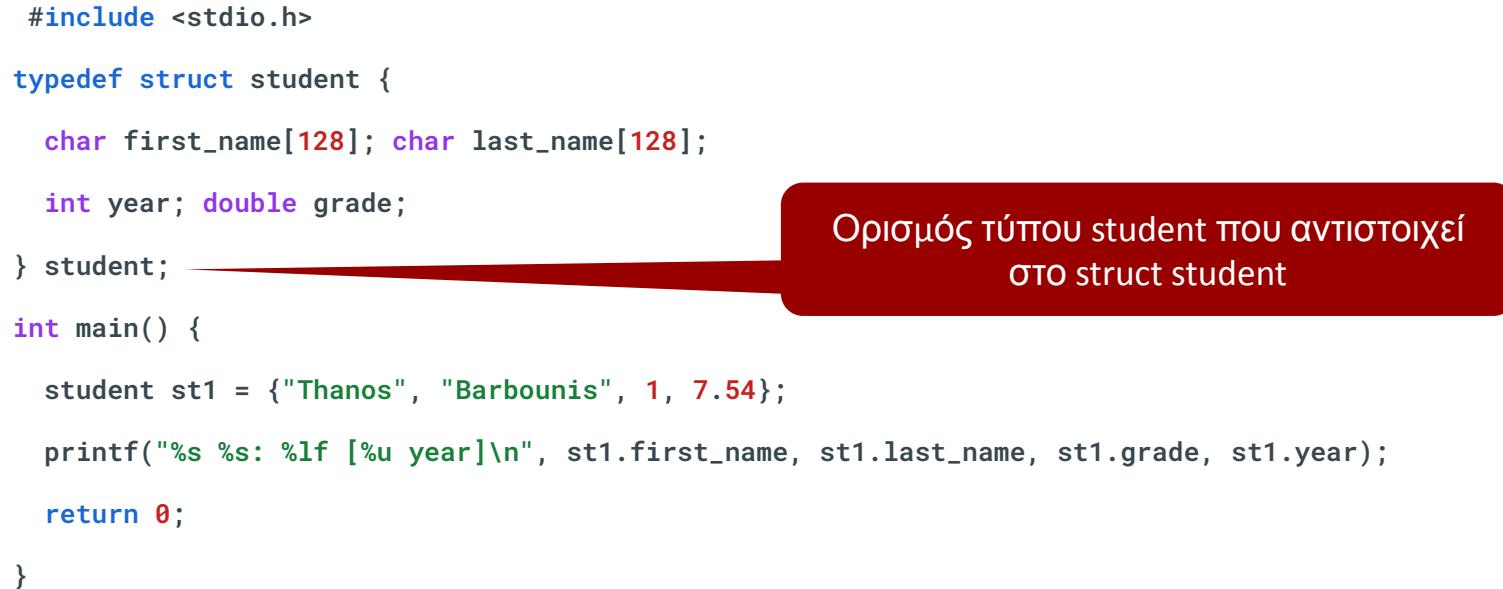
Structs και typedef

Μπορούμε να δημιουργήσουμε μια συντομογραφία student για τον τύπο struct student ως εξής:

```
#include <stdio.h>

typedef struct student {
    char first_name[128]; char last_name[128];
    int year; double grade;
} student;

int main() {
    student st1 = {"Thanos", "Barbounis", 1, 7.54};
    printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade, st1.year);
    return 0;
}
```



Ορισμός τύπου student που αντιστοιχεί στο struct student

Structs και typedef

Μπορούμε να δημιουργήσουμε μια συντομογραφία student για τον ανώνυμο τύπο struct ως εξής:

```
#include <stdio.h>

typedef struct {
    char first_name[128]; char last_name[128];
    int year; double grade;
} student;

int main() {
    student st1 = {"Thanos", "Barbounis", 1, 7.54};
    printf("%s %s: %lf [%u year]\n", st1.first_name, st1.last_name, st1.grade, st1.year);
    return 0;
}
```

Η ετικέτα του struct μπορεί να παραληφθεί

Εμφωλευμένες/Ένθετες Δομές (Nested Structs)

Μια δομή μπορεί να περιέχει **μία ή περισσότερες δομές**, οι οποίες ονομάζονται εμφωλευμένες/ένθετες δομές (nested structs). Γενική μορφή:

```
struct όνομα1 {  
    ...  
};  
  
struct όνομα2 {  
    ...  
    struct όνομα1 πεδίο3;  
    ...  
};
```

Η δομή όνομα1 είναι εμφωλευμένη στην δομή όνομα2

Εμφωλευμένες Δομές (Nested Structs)

```
#include <stdio.h>
int main() {
    struct date {
        int day;
        int month;
        int year;
    };
    struct product {
        char * name;
        double price;
        struct date created;
        struct date updated;
    };
    struct product prod = {"eclasse", 3.14, {1, 1, 2021}, {11, 12, 2022}};
    printf("%zu\n", sizeof(prod));
    return 0;
}
```

\$./nested
40

Η δομή date είναι εμφωλευμένη μέσα στην δομή product και χρησιμοποιείται από δύο πεδία

Εμφωλενμένα Πεδία Δομών (Nested Struct Fields)

```
#include <stdio.h>

struct date {
    int day;
    int month;
    int year;
};

struct product {
    char * name;
    double price;
    struct date created;
    struct date updated;
};

int main() {
    struct product prod = {"eclasse", 3.14, {1, 1, 2021}, {11, 12, 2022}};

    prod.updated.year = 2023;

    printf("%s [eu: %.2lf] [created: %d/%d/%d] [updated: %d/%d/%d]\n",
           prod.name, prod.price,
           prod.created.day, prod.created.month, prod.created.year,
           prod.updated.day, prod.updated.month, prod.updated.year);

    return 0;
}
```

Χρησιμοποιούμε
`.` όσες φορές
χρειαστεί για να
αναφερθούμε στο
πεδίο που θέλουμε

\$./nested2
eclasse [eu: 3.14] [created: 1/1/2021] [updated: 11/12/2023]

Εμφωλευμένα Πεδία Δομών (Nested Struct Fields)

```
#include <stdio.h>

typedef struct {
    int day;
    int month;
    int year;
} Date;

typedef struct {
    char * name;
    double price;
    Date created;
    Date updated;
} Product;

int main() {
    Product prod = {"eclasse", 3.14, {1, 1, 2021}, {11, 12, 2022}};
    prod.updated.year = 2023;
    printf("%s [eu: %.2lf] [created: %d/%d/%d] [updated: %d/%d/%d]\n",
           prod.name, prod.price,
           prod.created.day, prod.created.month, prod.created.year,
           prod.updated.day, prod.updated.month, prod.updated.year);
    return 0;
}

$ ./nested3
eclasse [eu: 3.14] [created: 1/1/2021] [updated: 11/12/2023]
```

Δείκτες σε Δομές

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

typedef struct { int day; int month; int year; } Date;

int main() {
    Date d1 = {1, 10, 2023};

    Date * d2 = &d1;
(*d2).day = 2;

    Date * d3 = malloc(sizeof(Date));
*d3 = *d2;

    (*d3).month = 12; (*d3).day = 11;

    printf("Diff: %d/%d/%d\n", (*d3).day - d1.day, (*d3).month - d1.month, (*d3).year - d1.year);

    return 0;
}
```

Οι δείκτες μπορούν να συνδυαστούν με δομές όπως όλοι οι άλλοι τύποι. Τι θα τυπώσει το διπλανό πρόγραμμα;

```
$ ./date
Diff: 9/2/0
```

Συντόμευση: Προσπέλαση Πεδίων Δομής μέσω Δείκτη (Arrow Operator)

Για να προσπελάσουμε το πεδίο μιας δομής, όταν έχουμε έναν δείκτη στην δομή χρησιμοποιούμε:

όνομα_μεταβλητής_τύπου_δείκτη_σε_δομή->όνομα_πεδίου

Για παράδειγμα:

```
student * st1;
```

```
...
```

```
st1->year
```

Αντί να γράφουμε `(*st1).year` μπορούμε να χρησιμοποιήσουμε την σύνταξη `st1->year`. Οι δύο εκφράσεις `(*ptr).field` και `ptr->field` είναι **ισοδύναμες**

Δείκτες σε Δομές με χρήση ->

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

typedef struct { int day; int month; int year; } Date;

int main() {
    Date d1 = {1, 10, 2023};

    Date * d2 = &d1;
    d2->day = 2;

    Date * d3 = malloc(sizeof(Date));
    *d3 = *d2;

    d3->month = 12; d3->day = 11;
    printf("Diff: %d/%d/%d\n", d3->day - d1.day, d3->month - d1.month, d3->year - d1.year);
    return 0;
}
```

Χρησιμοποιώντας -> γράφουμε έναν χαρακτήρα λιγότερο :) και υποδεικνύουμε στον αναγνώστη ότι η μεταβλητή είναι δείκτης

```
$ ./dateptr
Diff: 9/2/0
```

Διάλεξη 20 - Προχωρημένες Δομές

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Πεδία Δυνφίων σε Δομές (Struct Bit Fields)

Ένα πεδίο δομής μπορεί να οριστεί έτσι ώστε να έχει ως μέγεθος στην μνήμη ένα συγκεκριμένο αριθμό από bits. Συγκεκριμένα, ακολουθούμε την σύνταξη:

Γιατί; Για να σώσουμε μνήμη

```
struct όνομα {  
    τύπος1 πεδίο1 : αριθμός_bits1;  
    τύπος2 πεδίο2 : αριθμός_bits2;  
    τύπος3 πεδίο3 : αριθμός_bits3;  
    ...  
};
```

Τυπικά οι μεταγλωττιστές υποστηρίζουν τύπους int, long, char

Πεδία Δυνφίων σε Δομές (Struct Bit Fields)

Ένα πεδίο δομής μπορεί να οριστεί έτσι ώστε να έχει ως μέγεθος στην μνήμη ένα συγκεκριμένο αριθμό από bits. Συγκεκριμένα, ακολουθούμε την σύνταξη:

```
struct student_status {  
    int registered : 1;  
  
    int year : 3;  
  
    int grade : 4;  
};
```

Δύο καταστάσεις (boolean):
registered ή όχι - 1 bit αρκεί

Ο βαθμός είναι από το 0 μέχρι το 10, οπότε 4 bits φτάνουν για να αναπαραστήσουμε όλες τις δυνατές τιμές

Πεδία Δυνφίων σε Δομές (Struct Bit Fields)

Ένα πεδίο δομής μπορεί να οριστεί έτσι ώστε να έχει ως μέγεθος στην μνήμη ένα συγκεκριμένο αριθμό από bits. Συγκεκριμένα, ακολουθούμε την σύνταξη:

```
struct student_status {  
    int registered : 1;  
    int year : 3;  
    int grade : 4;  
};  
printf("%zu\n", sizeof(struct student_status));
```

Πεδία Δυνφίων σε Δομές (Struct Bit Fields)

Ένα πεδίο δομής μπορεί να οριστεί έτσι ώστε να έχει ως μέγεθος στην μνήμη ένα συγκεκριμένο αριθμό από bits. Συγκεκριμένα, ακολουθούμε την σύνταξη:

```
struct student_status {  
    int registered : 1;  
    int year : 3;                      $ ./bitfields  
                                         4  
    int grade : 4;  
};  
printf("%zu\n", sizeof(struct student_status));
```

Πεδία Δυνφίων σε Δομές (Struct Bit Fields)

Ένα πεδίο δομής μπορεί να οριστεί έτσι ώστε να έχει ως μέγεθος στην μνήμη ένα συγκεκριμένο αριθμό από bits. Συγκεκριμένα, ακολουθούμε την σύνταξη:

```
struct student_status {  
    char registered : 1;  
    char year : 3;                      $ ./bitfields2  
                                         1  
    char grade : 4;  
};  
printf("%zu\n", sizeof(struct student_status));
```

Bit Fields - Αναπαράσταση

```
struct student_status {  
    char registered;  
    char year;  
    char grade;  
};  
  
printf("%zu\n", sizeof(struct student_status));
```

```
$ ./bitfields2  
3
```

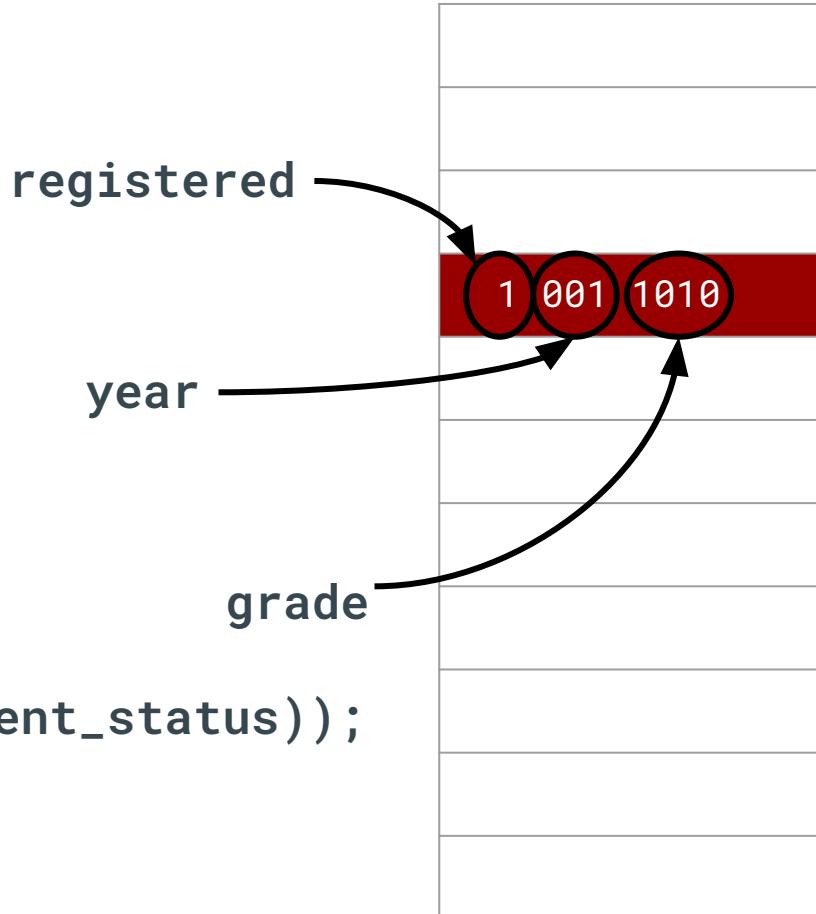
registered
year
grade

byte
byte
byte

Bit Fields - Αναπαράσταση

```
struct student_status {  
    char registered : 1;  
    char year : 3;  
    char grade : 4;  
};  
  
printf("%zu\n", sizeof(struct student_status));
```

```
$ ./bitfields2  
1
```



Bit Fields - Ανάθεση

```
#include <stdio.h>

typedef struct {

    unsigned char registered : 1;
    unsigned char year : 3;
    unsigned char grade : 4;
} status;

int main() {

    status st = {1, 1, 10};

    printf("Status: %u %u %u\n", st.registered, st.year, st.grade);

    st.year = 2;

    printf("Status: %u %u %u\n", st.registered, st.year, st.grade);

    st.year += 7;

    printf("Status: %u %u %u\n", st.registered, st.year, st.grade);

    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

```
$ ./bitfields3
Status: 1 1 10
Status: 1 2 10
Status: 1 1 10
```

$$2 + 7 = 1??$$



Bit Fields - Ανάθεση

```
#include <stdio.h>

typedef struct {

    unsigned char registered : 1;
    unsigned char year : 3;
    unsigned char grade : 4;
} status;

int main() {

    status st = {1, 1, 10};

    printf("Status: %u %u %u\n", st.registered, st.year, st.grade);

    st.year = 2;
    printf("Status: %u %u %u\n", st.registered, st.year, st.grade);
    st.year += 7;
    printf("Status: %u %u %u\n", st.registered, st.year, st.grade);

    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

```
$ ./bitfields3
Status: 1 1 10
Status: 1 2 10
Status: 1 1 10
```

$$2 + 7 \% 8 = 1 \% 8$$

Το **year** έχει μόλις **3 bits** - αν προσπαθήσουμε να αποθηκεύσουμε μεγαλύτερες τιμές του $7 (2^3 - 1)$ θα υποστούμε τις συνέπειες (aka Προσοχή)

Περιορισμοί Bit Fields

1. Δεν μπορούμε να ζητήσουμε το μέγεθος (sizeof αποτυγχάνει)

```
bitfields.c:14:25: error: 'sizeof' applied to a bit-field  
14 |     printf("%zu\n", sizeof(st.year));
```

2. Δεν μπορούμε να πάρουμε την διεύθυνση ενός bit-field

```
bitfields.c:14:23: error: cannot take address of bit-field 'year'  
14 |     unsigned char * c = &st.year;
```

3. Μπορεί να οδηγήσει σε προβλήματα απόδοσης, δυσκολία ανάγνωσης, ή προβλήματα συμβατότητας. Αποφεύγουμε ή κάνουμε χρήση με φειδώ.

Ενώσεις (Unions)

Η **ένωση** (union) στην C μοιάζει με μία δομή (struct) με μία διαφορά: τα πεδία της ένωσης μοιράζονται την **ίδια μνήμη**. Η κοινή μνήμη επιτρέπει **εξοικονόμηση χώρου**.

Για μία μεταβλητή τύπου union επομένως, συνήθως έχει νόημα να χρησιμοποιούμε **μόνο ένα** από τα πεδία της.

Δήλωση Ένωσης (Union Declaration)

Η ένωση (union) στην C μοιάζει με μία δομή (struct) με μία διαφορά: τα πεδία της ένωσης μοιράζονται την **ίδια μνήμη**. Γενική σύνταξη:

```
union όνομα {  
    τύπος1 πεδίο1;  
    τύπος2 πεδίο2;  
    τύπος3 πεδίο3;  
    ...  
};
```

Όμοια με την δήλωση struct απλά χρησιμοποιεί το keyword union

Δήλωση Ένωσης (Union Declaration)

Η ένωση (union) στην C μοιάζει με μία δομή (struct) με μία διαφορά: τα πεδία της ένωσης μοιράζονται την **ίδια μνήμη**. Γενική σύνταξη:

```
union anything {  
    char c;  
    int i;  
    float f;  
    double d;  
};
```

Χρήση Ένωσης και Μέγεθος

```
#include <stdio.h>

union anything {
    char c;  int i;  float f;  double d;
};

int main() {
    union anything a1;
    a1.i = 0x42;
    printf("%d %c\n", a1.i, a1.c);
    a1.c = 'C';
    printf("%d %c\n", a1.i, a1.c);
    printf("%zu\n", sizeof(a1));
    return 0;
}
```

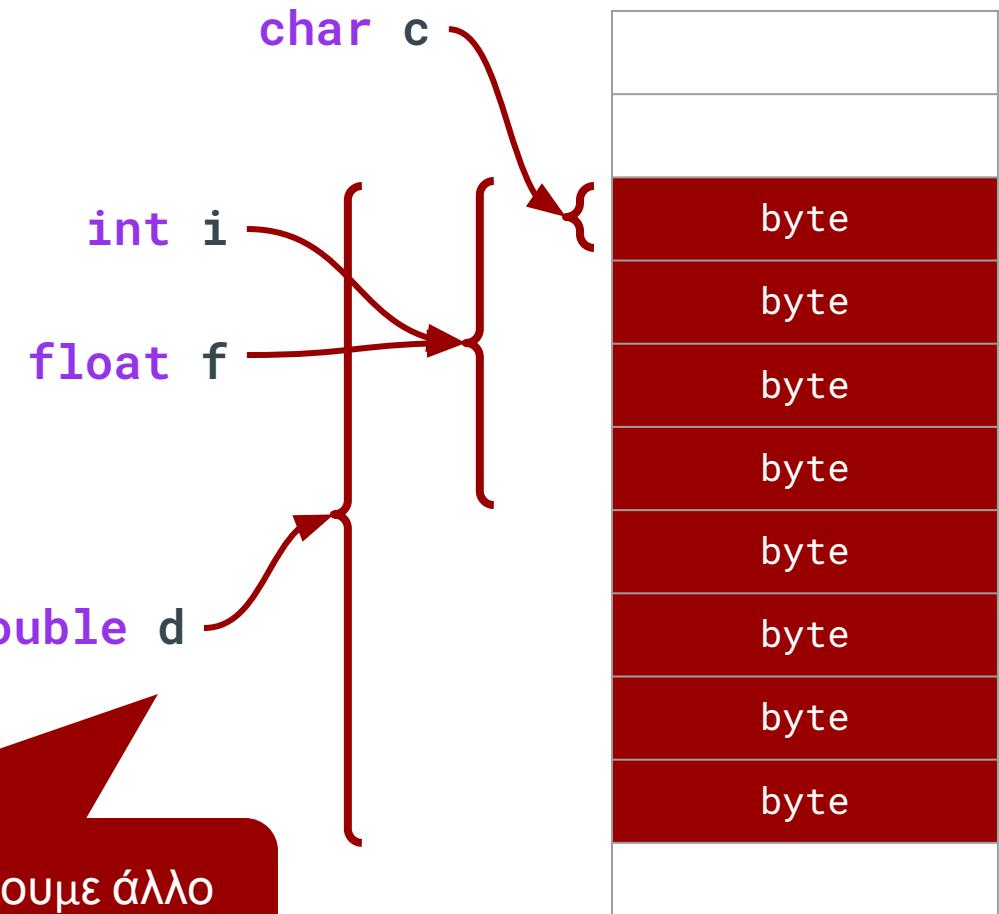
Τι θα τυπώσει το πρόγραμμα;

```
$ ./union
66 B
67 C
8
```

Χρήση Ένωσης και Μέγεθος

```
#include <stdio.h>
union anything {
    char c;  int i;  float f;  double d;
};
int main() {
    union anything a1;
    a1.i = 0x42;
    printf("%d %c\n", a1.i, a1.c);
    a1.c = 'C';
    printf("%d %c\n", a1.i, a1.c);
}
```

```
$ ./union
66 B
67 C
8
```

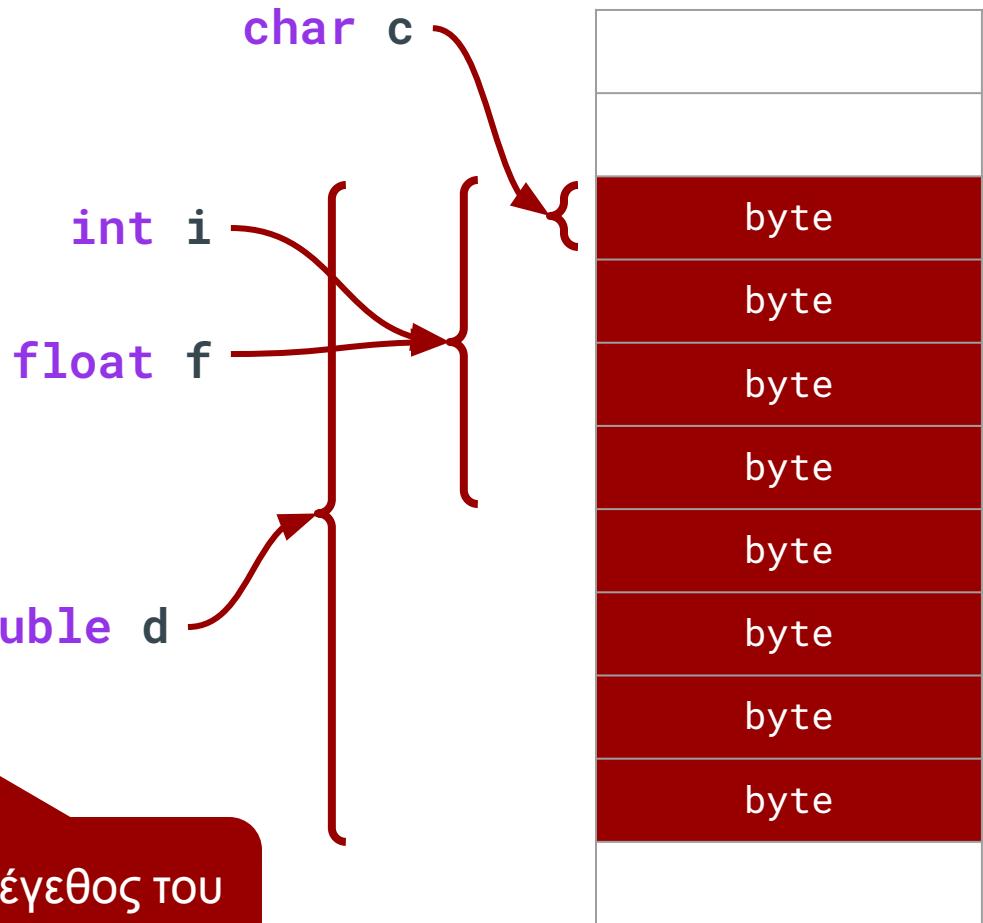


Διαλέγοντας διαφορετικό πεδίο, διαλέγουμε άλλο τρόπο αναπαράστασης των ίδιων bytes στην μνήμη

Χρήση Ένωσης και Μέγεθος

```
#include <stdio.h>
union anything {
    char c;  int i;  float f;  double d;
};
int main() {
    union anything a1;
    a1.i = 0x42;
    printf("%d %c\n", a1.i, a1.c);
    a1.c = 'C';
    printf("%d %c\n", a1.i, a1.c);
}
```

```
$ ./union
66 B
67 C
8
```



Το μέγεθος της ένωσης ταυτίζεται με το μέγεθος του τύπου του "μεγαλύτερου" πεδίου της

Απαριθμήσεις (Enumerations)

Ο τύπος απαριθμησης enum (enumeration type) μας επιτρέπει να ορίζουμε ένα σύνολο ακεραίων με συγκεκριμένα ονόματα και σταθερές τιμές.

Η πρώτη σταθερά αρχικοποιείται στο 0 (εκτός αν της δοθεί συγκεκριμένη τιμή).

Κάθε σταθερά στην οποία δεν δίνουμε τιμή παίρνει την τιμή της προηγούμενης σταθεράς αυξημένη κατά 1.

Δήλωση Απαρίθμησης (Enum Declaration)

Ο τύπος απαρίθμησης enum (enumeration type) μας επιτρέπει να ορίζουμε ένα σύνολο ακεραίων με συγκεκριμένα ονόματα και σταθερές τιμές. Σύνταξη:

enum όνομα { επιλογή1, επιλογή2, ... };

Όνομα της απαρίθμησης

Σταθερές που αποτελούν
την απαρίθμηση

Παράδειγμα:

enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

Χρήση Απαρίθμησης (Enum Usage)

Τι θα τυπώσει το ακόλουθο πρόγραμμα;

```
#include <stdio.h>

enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

int main() {
    enum weekday day1 = Mon, day2;
    day2 = Fri;
    printf("%d %d\n", day1, day2);
    return 0;
}
```

\$./enum1
0 4

Χρήση Απαρίθμησης (Enum Usage) #2

Τι θα τυπώσει το ακόλουθο πρόγραμμα;

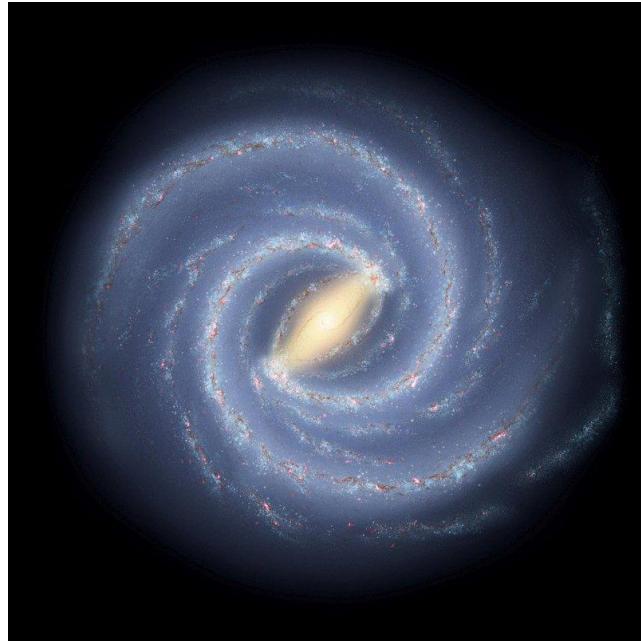
```
#include <stdio.h>

enum weekday {Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun};

int main() {
    for(enum weekday day = Mon; day <= Sun; day++) {
        if(day == Mon || day == Fri)
            printf("We have class on day # %d of the week\n", day);
    }
    return 0;
}
```

\$./enum2
We have class on day # 1 of the week
We have class on day # 5 of the week

Αυτοαναφορά (Self-Reference)

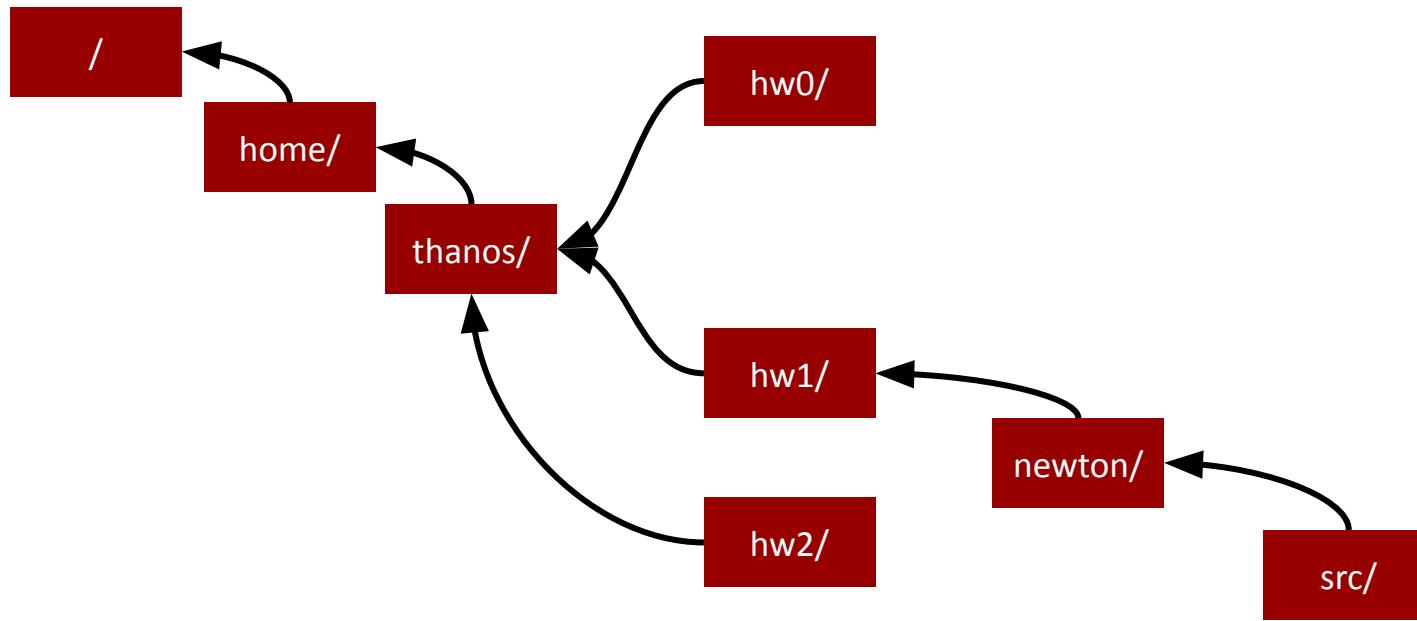


Αυτή η πρόταση είναι ψευδής

(Επιμενίδης, 6^{ος} αιώνας π.Χ.)

Θέλω να φτιάξω μια δομή (struct) που να αναπαριστά έναν φάκελο σε ένα σύστημα αρχείων. Πως θα το κάνω;

Κάθε φάκελος έχει ένα όνομα και βρίσκεται μέσα σε κάποιον άλλο (parent) φάκελο. Ο αρχικός φάκελος (root /) δεν βρίσκεται μέσα σε κάποιο φάκελο. #1 εκδοχή:



Αυτοαναφορικές Δομές (Self-Referential Structs)

Τα μέλη μιας δομής μπορεί να είναι οποιουδήποτε τύπου, ακόμα και **δείκτες του ίδιου τύπου**. Για παράδειγμα:

```
struct folder {  
    char name[128];  
    struct folder * parent;  
};
```

Κάθε φάκελος
έχει ένα όνομα

Κάθε φάκελος έχει
ένα δείκτη στον
parent φάκελο

Χρήση Αυτοαναφορικών Δομών

```
#include <stdio.h>

typedef struct folder {
    char name[128];
    struct folder * parent;
} Folder;

int main() {
    Folder root = {"/", NULL};

    Folder home = {"home/", &root};
    Folder user = {"thanos/", &home};

    Folder hw0 = {"hw0/", &user}, hw1 = {"hw1/", &user};

    Folder newton = {"newton/", &hw0};

    printf("%s -> %s -> %s\n", newton.name, newton.parent->name, newton.parent->parent->name);

    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

```
$ ./self
newton/ -> hw0/ -> thanos/
```

Χρήση Αυτοαναφορικών Δομών #2

```
#include <stdio.h>

typedef struct folder {
    char name[128];
    struct folder * parent;
} Folder;

int main() {
    Folder root = {"/", NULL};
    Folder home = {"home/", &root};

    Folder user = {"thanos/", &home};

    Folder hw0 = {"hw0/", &user}, hw1 = {"hw1/", &user};

    Folder newton = {"newton/", &hw0};

    for(Folder * iterator = &newton; iterator; iterator = iterator->parent)
        printf("folder: %s\n", iterator->name);

    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

```
$ ./self2
folder: newton/
folder: hw0/
folder: thanos/
folder: home/
folder: /
```

Αυτοαναφορικές Δομές - Μνήμη

```
#include <stdio.h>

typedef struct folder {
    char name[128];
    struct folder * parent;
} Folder;

int main() {
    Folder root = {"/", NULL};

    Folder home = {"home/", &root};

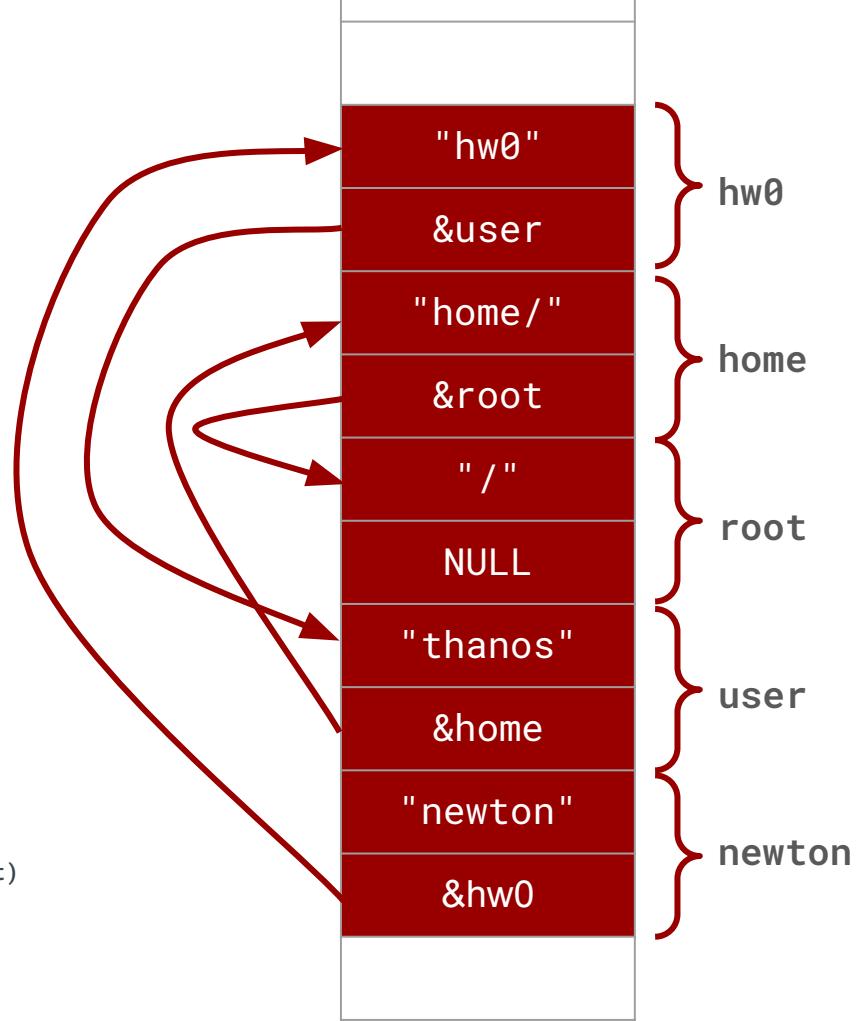
    Folder user = {"thanos/", &home};

    Folder hw0 = {"hw0/", &user}, hw1 = {"hw1/", &user};

    Folder newton = {"newton/", &hw0};

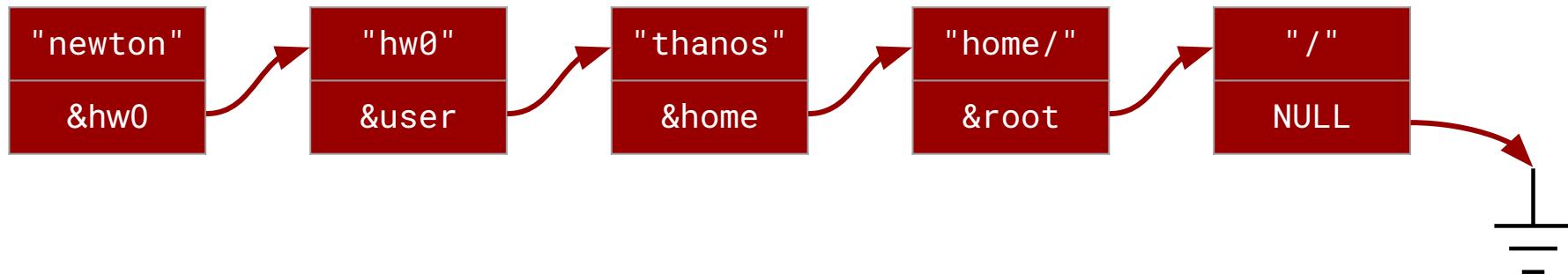
    for(Folder * iterator = &newton; iterator; iterator = iterator->parent)
        printf("folder: %s\n", iterator->name);

    return 0;
}
```



Σε πιο αφηρημένη (abstract) μορφή

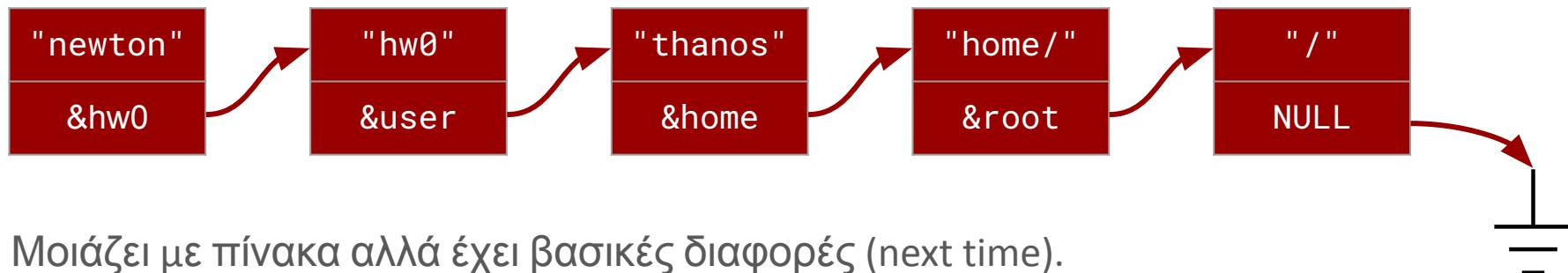
Η πραγματική διάταξη (layout) στην μνήμη μπορεί να είναι περίπλοκη. Σε αφηρημένη μορφή είναι κάπως έτσι:



Μας θυμίζει κάτι αυτή η διάταξη;

Απλά Συνδεδεμένη Λίστα (Single Linked List)

Η απλά συνδεδεμένη λίστα (single linked list) είναι ένας τύπος δεδομένων που το κάθε στοιχείο δείχνει στο επόμενο και το τελευταίο δείχνει στο NULL.



Μοιάζει με πίνακα αλλά έχει βασικές διαφορές (next time).

Συνήθως αποθηκεύεται εξ' ολοκλήρου δυναμικά (στον σωρό).

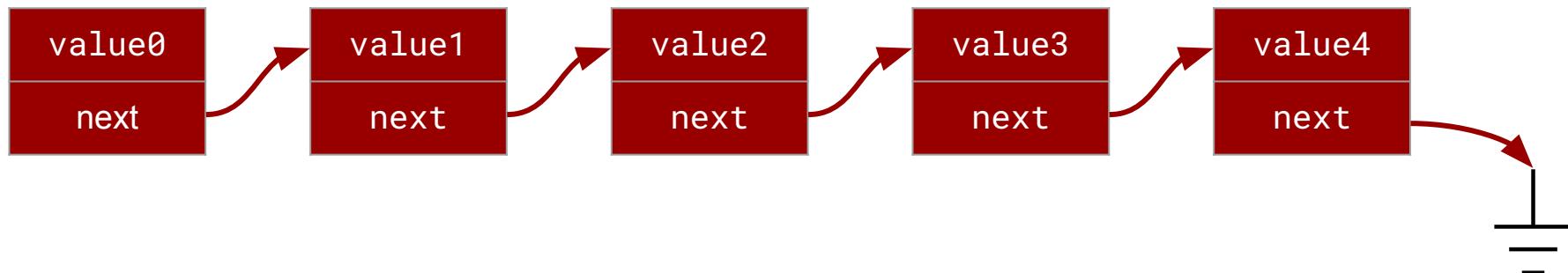
Απλά Συνδεδεμένη Λίστα (Single Linked List)

Στην γενική μορφή δηλώνεται ως:

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

Απλά Συνδεδεμένη Λίστα (Single Linked List)

Η απλά συνδεδεμένη λίστα (single linked list) είναι ένας τύπος δεδομένων που το κάθε στοιχείο δείχνει (links) στο επόμενο και το τελευταίο δείχνει στο NULL.



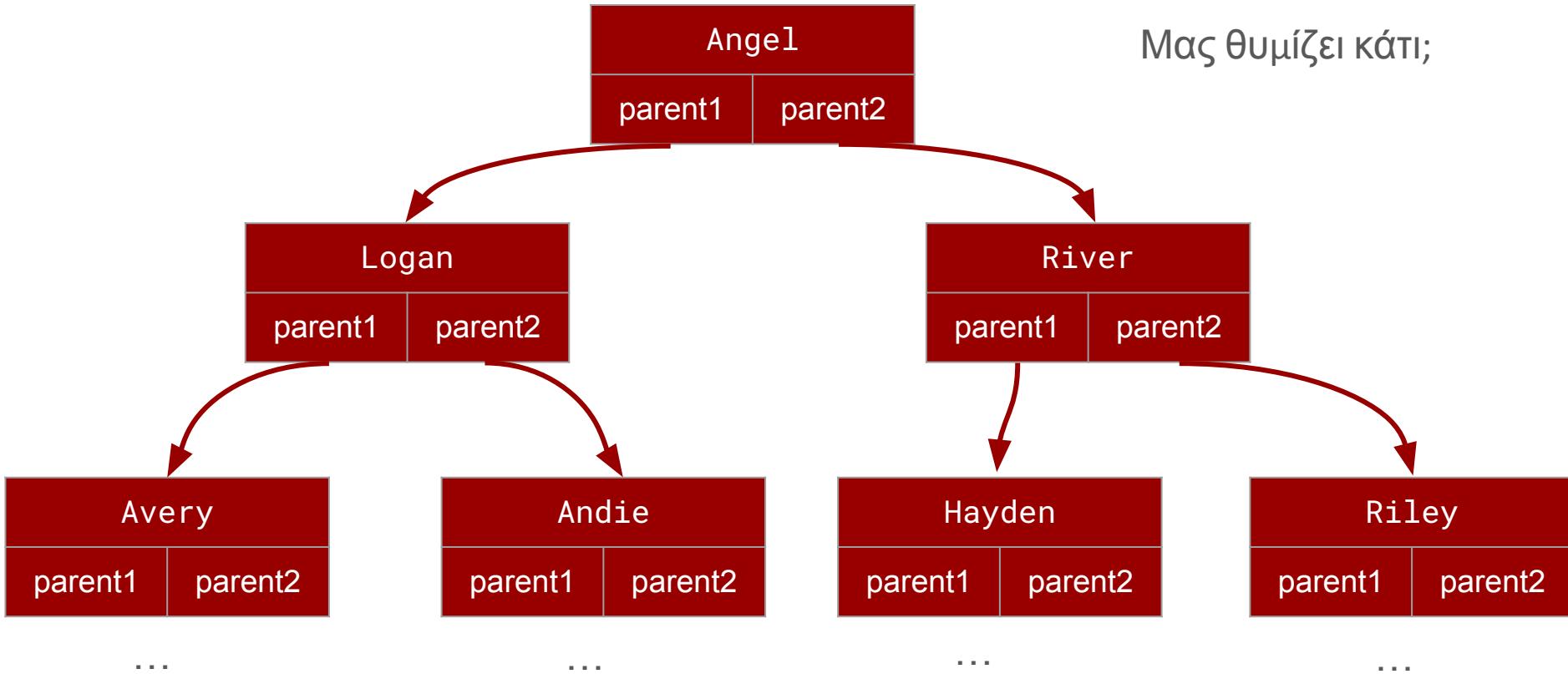
Μήκος λίστας (list length): ο αριθμός των στοιχείων που περιέχει (5 παραπάνω)

Αυτοαναφορικές Δομές (Self-Referential Structs)

Τα μέλη μιας δομής μπορεί να είναι οποιουδήποτε τύπου, ακόμα και **πολλοί δείκτες** του ίδιου τύπου. Για παράδειγμα:

```
struct person {  
    char name[128];  
    struct person * parent1;  
    struct person * parent2;  
};
```

Αυτοαναφορικές Δομές (Self-Referential Structs)



Δυαδικό Δέντρο (Binary Tree)

Το **δυαδικό δέντρο** (binary tree) είναι ένας τύπος δεδομένων που μας επιτρέπει να οργανώνουμε τα δεδομένα μας σε δενδρική διάταξη, καθένας από τους κόμβους του δέντρου μπορεί να έχει από 0 μέχρι 2 κόμβους-παιδιά.

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

Εφαρμογές: από βάσεις δεδομένων/αναζήτηση μέχρι μεταγλωττιστές και από συμπίεση δεδομένων μέχρι κρυπτογραφία (όπου απαιτείται αναπαράσταση γνώσης)

Δυαδικά Δέντρα (Binary Trees)

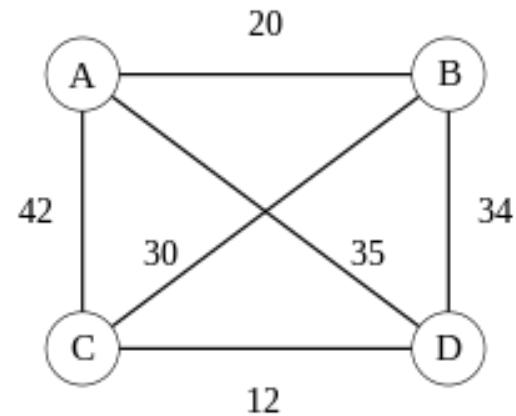
Ρίζα (root) του δέντρου (ο πρώτος κόμβος)



Φύλλα (leaves) του δέντρου (κόμβοι χωρίς παιδιά)

Βάθος (depth) του δέντρου (μέγιστος αριθμός συνδέσμων από την ρίζα μέχρι τα φύλλα) - εδώ 2

Θέλω να αναπαραστήσω ένα χάρτη σε μορφή γράφου με αυτοαναφορικές δομές. Πως;



Θέλω να αναπαραστήσω ένα σύστημα αρχείων (filesystem) ώστε να βρίσκω άμεσα τους υποφακέλους και τον parent φάκελο. Πως;

Διάλεξη 21 - Λίστες και Δέντρα

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

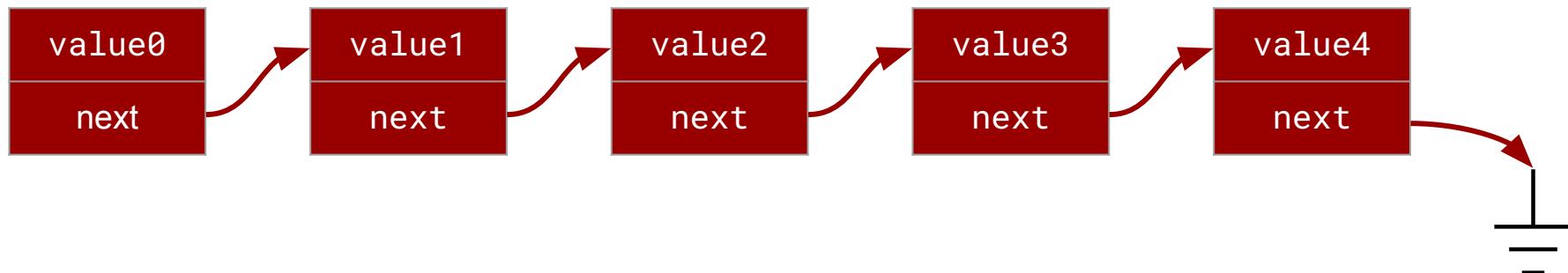
Απλά Συνδεδεμένη Λίστα (Single Linked List)

Στην γενική μορφή δηλώνεται ως:

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

Απλά Συνδεδεμένη Λίστα (Single Linked List)

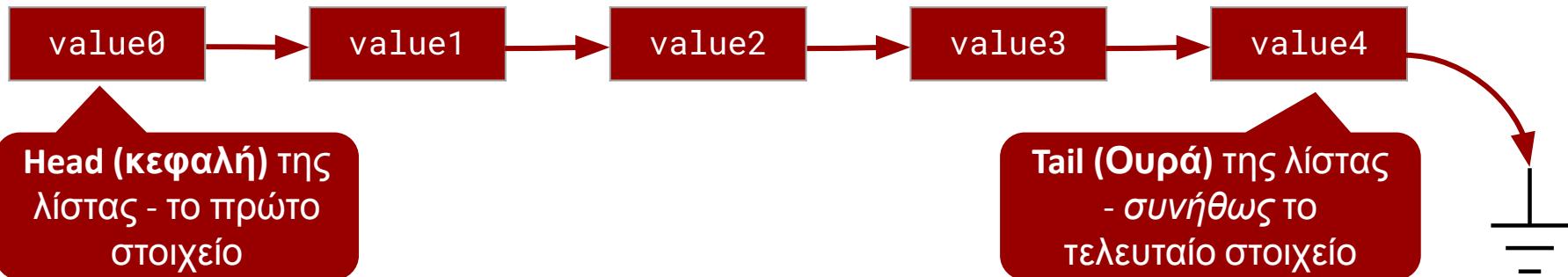
Η απλά συνδεδεμένη λίστα (single linked list) είναι ένας τύπος δεδομένων που το κάθε στοιχείο δείχνει (links) στο επόμενο και το τελευταίο δείχνει στο NULL.



Μήκος λίστας (list length): ο αριθμός των στοιχείων που περιέχει (5 παραπάνω)

Απλά Συνδεδεμένη Λίστα (Single Linked List)

Η απλά συνδεδεμένη λίστα (single linked list) είναι ένας τύπος δεδομένων που το κάθε στοιχείο δείχνει (links) στο επόμενο και το τελευταίο δείχνει στο NULL.



Μήκος λίστας (list length): ο αριθμός των στοιχείων που περιέχει (5 παραπάνω)

Βασικές Λειτουργίες με Λίστες

1. is_empty: Έλεγχος αν η λίστα είναι άδεια
2. insert: Προσθήκη στοιχείου στην λίστα
3. print: Τύπωμα στοιχείων λίστας
4. length: Εύρεση μήκους λίστας
5. find: Εύρεση στοιχείου σε λίστα
6. delete: Αφαίρεση στοιχείου από λίστα

is_empty: Έλεγχος αν η λίστα είναι άδεια

```
#include <stdio.h>

typedef struct listnode {int value; struct listnode * next;} * List;

int is_empty(List list) {
    return list == NULL;
}

int main() {
    struct listnode node = {42, NULL};

    List list1 = &node;
    List list2 = NULL;

    printf("Is empty: %d\n", is_empty(list1));
    printf("Is empty: %d\n", is_empty(list2));

    return 0;
}
```

is_empty: Έλεγχος αν η λίστα είναι άδεια

```
#include <stdio.h>

typedef struct listnode {int value; struct listnode * next;} * List;

int is_empty(List list) {
    return list == NULL;
}

int main() {
    struct listnode node = {42, NULL};

    List list1 = &node;
    List list2 = NULL;

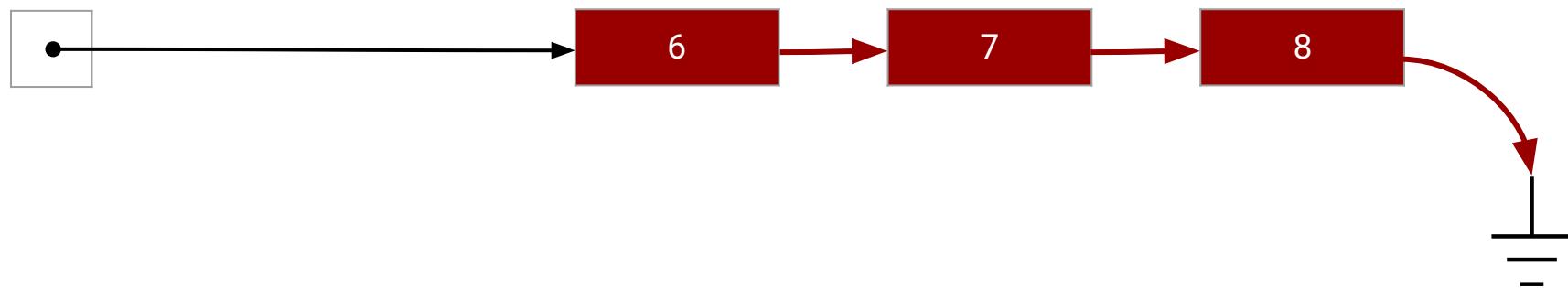
    printf("Is empty: %d\n", is_empty(list1));
    printf("Is empty: %d\n", is_empty(list2));

    return 0;
}
```

\$./list
Is empty: 0
Is empty: 1

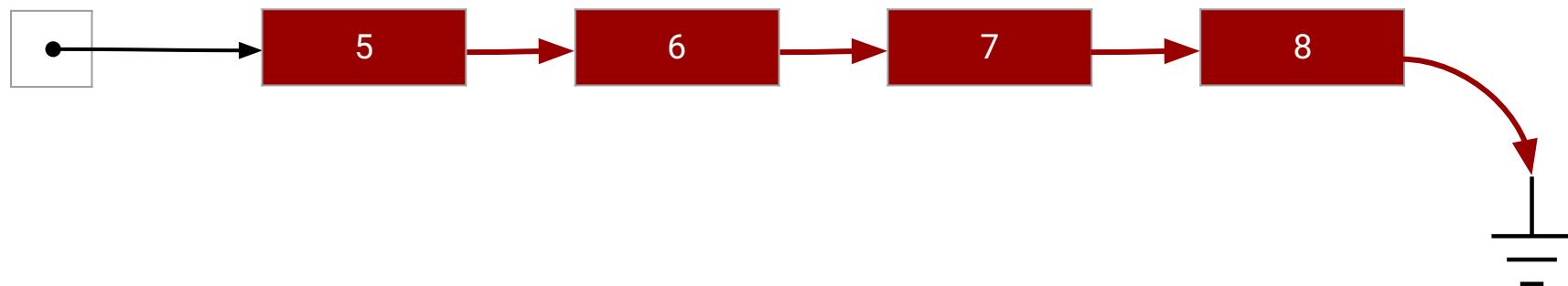
Θέλω να προσθέσω ένα στοιχείο (π.χ., το 5) σε λίστα. Πως;

list



Θέλω να προσθέσω ένα στοιχείο (π.χ., το 5) σε λίστα. Πως;

list



insert: Προσθήκη στοιχείου στην λίστα

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

typedef struct listnode {int value; struct listnode * next;} * List;

void insert(List * list, int value) {
    List current_head = *list;
    List new_head = malloc(sizeof(struct listnode));
    new_head->value = value;
    new_head->next = current_head;
    *list = new_head;
}

int main() {
    List list = NULL;
    insert(&list, 42); insert(&list, 43); insert(&list, 44);
    print(list);
    return 0;
}
```

Δημιουργία νέου
κόμβου λίστας
στον σωρό (heap)

Αρχικοποίηση
κόμβου

Ο νέος κόμβος γίνεται η νέα
κεφαλή της λίστας

insert: Προσθήκη στοιχείου στην λίστα

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

typedef struct listnode {int value; struct listnode * next;} * List;

void insert(List * list, int value) {
    List current_head = *list;
    List new_head = malloc(sizeof(struct listnode));
    new_head->value = value;
    new_head->next = current_head;
    *list = new_head;
}

int main() {
    List list = NULL;
    insert(&list, 42); insert(&list, 43); insert(&list, 44);
    print(list); // commented out for size
    return 0;
}
```

Τι θα τυπώσει το πρόγραμμα;

```
$ ./insert
list: -> 44 -> 43 -> 42 -> NULL
```

print: Τύπωμα στοιχείων λίστας

```
void print(List list) {  
    printf("list: ");  
    while(list) {  
        printf(" -> %d", list->value);  
        list = list->next;  
    }  
    printf(" -> NULL\n");  
}
```

length: Εύρεση μήκους λίστας

```
int length(List list) {  
    int counter = 0;  
    while(list) {  
        counter++;  
        list = list->next;  
    }  
    return counter;  
}
```

Πως θα το κάναμε αναδρομικά;

length: Εύρεση μήκους λίστας

```
int length(List list) {  
    int counter = 0;  
    while(list) {  
        counter++;  
        list = list->next;  
    }  
    return counter;  
}
```

```
int length(List list) {  
    if (!list) return 0;  
    return 1 + length(list->next);  
}
```

Ποια η πολυπλοκότητα των παραπάνω
ως προς χρόνο και χώρο;

length: Εύρεση μήκους λίστας

```
int length(List list) {  
    int counter = 0;  
    while(list) {  
        counter++;  
        list = list->next;  
    }  
    return counter;  
}
```

Χρόνος: $O(n)$
Χώρος: $O(1)$

```
int length(List list) {  
    if (!list) return 0;  
    return 1 + length(list->next);  
}
```

Ποια η πολυπλοκότητα των παραπάνω
ως προς χρόνο και χώρο;

Χρόνος: $O(n)$
Χώρος: $O(n)$

find: Εύρεση στοιχείου σε λίστα

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

typedef struct listnode {int value; struct listnode * next;} * List;

List find(List list, int value) {
    while(list && list->value != value) {
        list = list->next;
    }
    return list;
}

int main() {
    List list = NULL;
    insert(&list, 42); insert(&list, 43); insert(&list, 44);
    printf("Found 43: %x\n", find(list, 43));
    printf("Found 34: %x\n", find(list, 34));
    return 0;
}
```

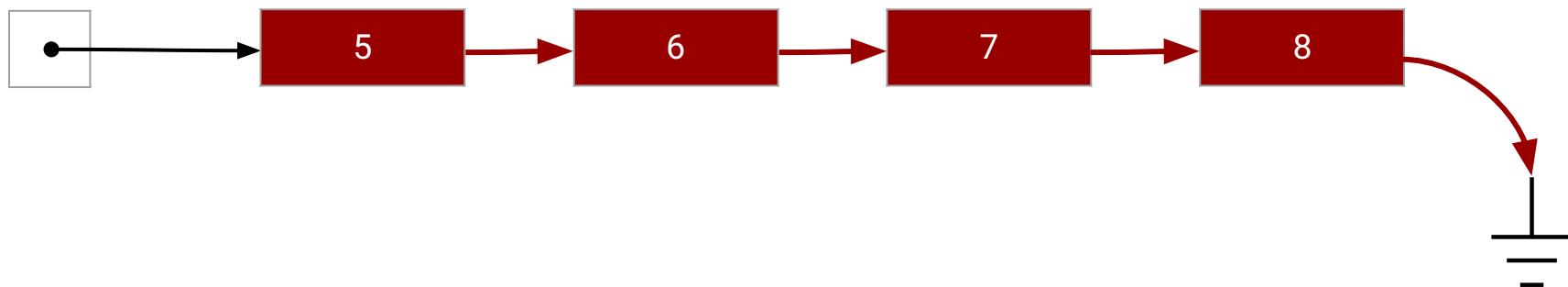
Τι θα τυπώσει το πρόγραμμα;

```
$ ./find
Found 43: 161862c0
Found 34: 0
```

Χρόνος: $O(n)$
Χώρος: $O(1)$

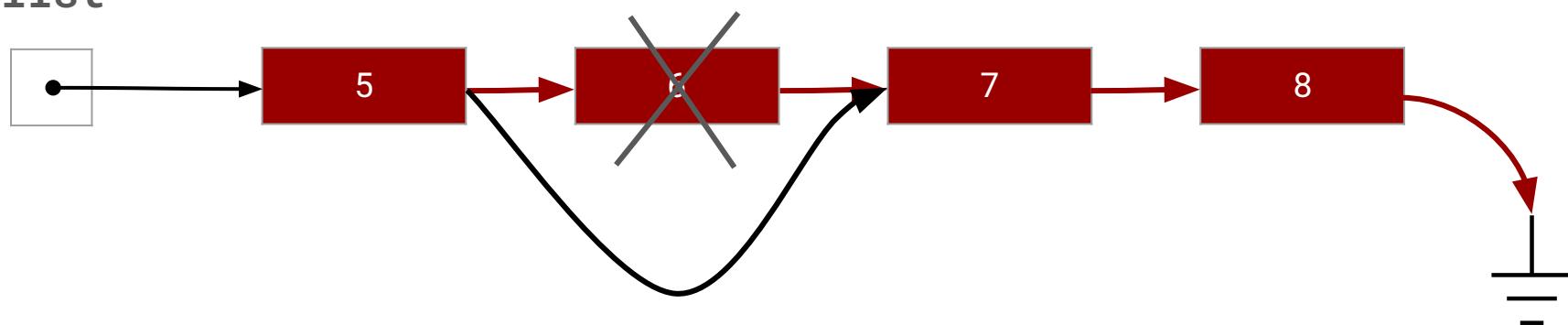
Θέλω να αφαιρέσω ένα στοιχείο (π.χ., το 6). Πως;

list



Θέλω να αφαιρέσω ένα στοιχείο (π.χ., το 6). Πως;

list



delete: Αφαίρεση στοιχείου από λίστα

```
void delete(List * list, int value) {  
    List temp;  
  
    while(*list && (*list)->value != value) {  
        list = &((*list)->next);  
    }  
  
    if (*list) {  
        temp = *list;  
        *list = temp->next;  
        free(temp);  
    }  
}
```

Πίνακες vs Λίστες

Πίνακες

1. Τα στοιχεία αποθηκεύονται σε **συνεχόμενες θέσεις** στην μνήμη
2. Ο χώρος μνήμης είναι **όσος χρειάζεται** για την αποθήκευση των στοιχείων
3. **Πρόσβαση** σε κάθε στοιχείο **σε σταθερό χρόνο** ($O(1)$) χρησιμοποιώντας $array[i]$
4. **Αναδιάταξη** στοιχείων συνήθως παίρνει **$O(n)$ χρόνο** (π.χ., εισαγωγή στο $array[0]$)
5. Στην **δήλωσή** τους χρειάζεται να **ξέρουμε πόσα στοιχεία** θα εισάγουμε (πιο στατικό ως δομή δεδομένων)

Λίστες

1. Τα στοιχεία μπορούν να αποθηκευτούν **σε οποιαδήποτε θέση** στην μνήμη
2. Ο χώρος μνήμης είναι **επαυξημένος** κατά **ένα sizeof(pointer)** για κάθε στοιχείο
3. **Πρόσβαση** σε κάθε στοιχείο **σε γραμμικό χρόνο** ($O(n)$)
4. Εύκολη / γρήγορη **αναδιάταξη** στοιχείων με **χρήση δεικτών**
5. Στην **δήλωσή** τους δεν χρειάζεται να **ξέρουμε πόσα στοιχεία** θα εισάγουμε (πιο δυναμικό ως δομή δεδομένων)



Δυαδικό Δέντρο (Binary Tree)

Το **δυαδικό δέντρο** (binary tree) είναι ένας τύπος δεδομένων που μας επιτρέπει να οργανώνουμε τα δεδομένα μας σε δενδρική διάταξη, καθένας από τους κόμβους του δέντρου μπορεί να έχει από 0 μέχρι 2 κόμβους-παιδιά.

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

Εφαρμογές: από βάσεις δεδομένων/αναζήτηση μέχρι μεταγλωττιστές και από συμπίεση δεδομένων μέχρι κρυπτογραφία (όπου απαιτείται αναπαράσταση γνώσης)

Δυαδικά Δέντρα (Binary Trees)

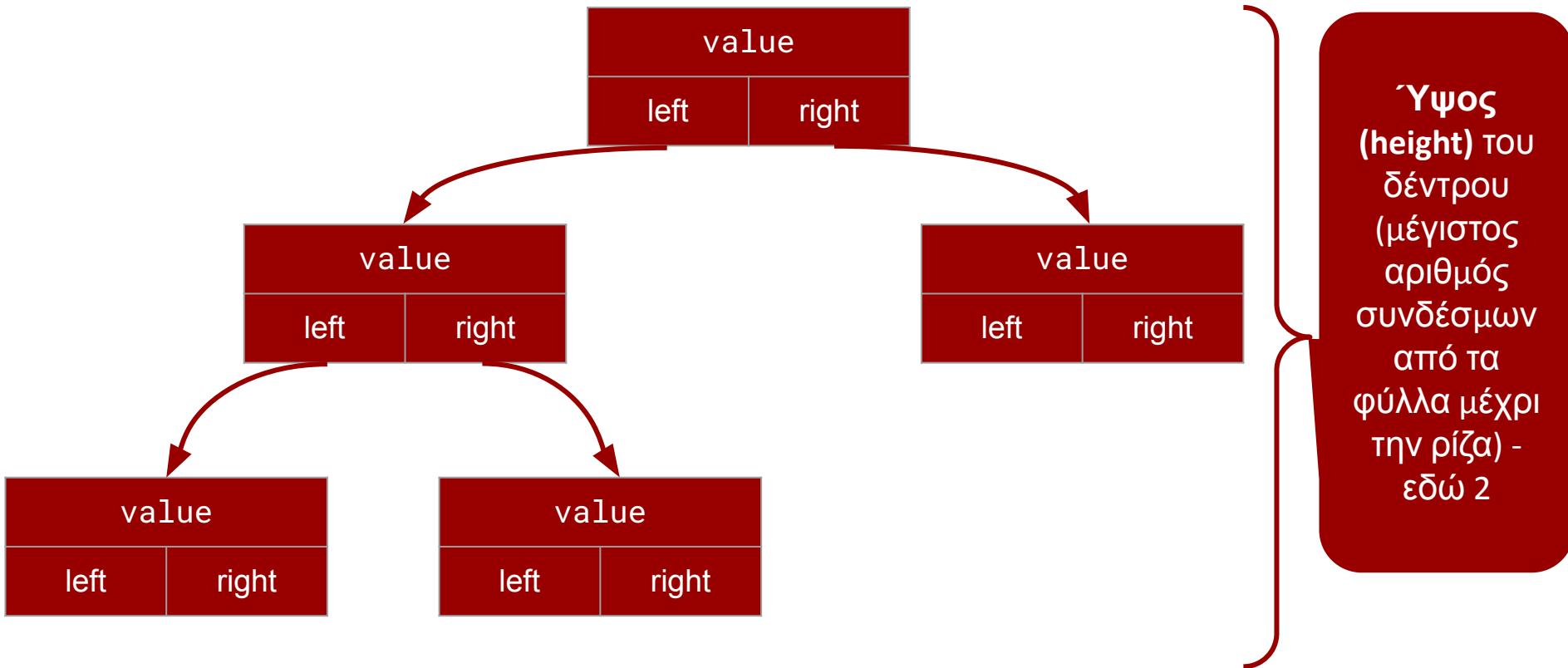
Ρίζα (root) του δέντρου (ο πρώτος κόμβος)



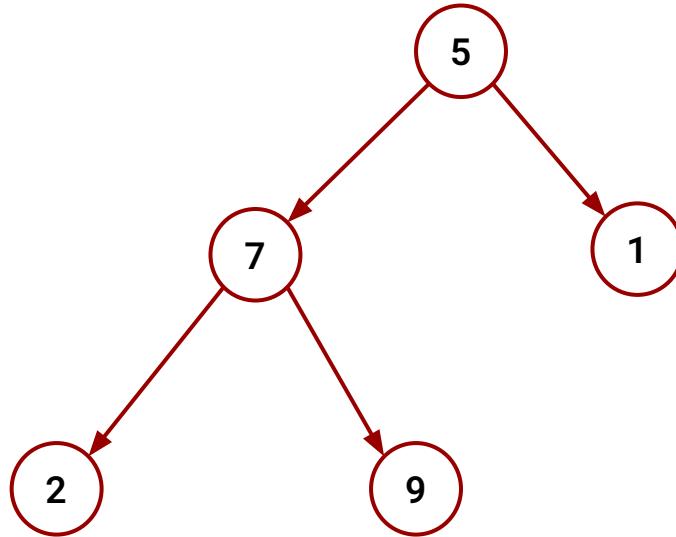
Φύλλα (leaves) του δέντρου (κόμβοι χωρίς παιδιά)

Βάθος (depth) του δέντρου (μέγιστος αριθμός συνδέσμων από την ρίζα μέχρι τα φύλλα) - εδώ 2

Δυαδικά Δέντρα (Binary Trees)

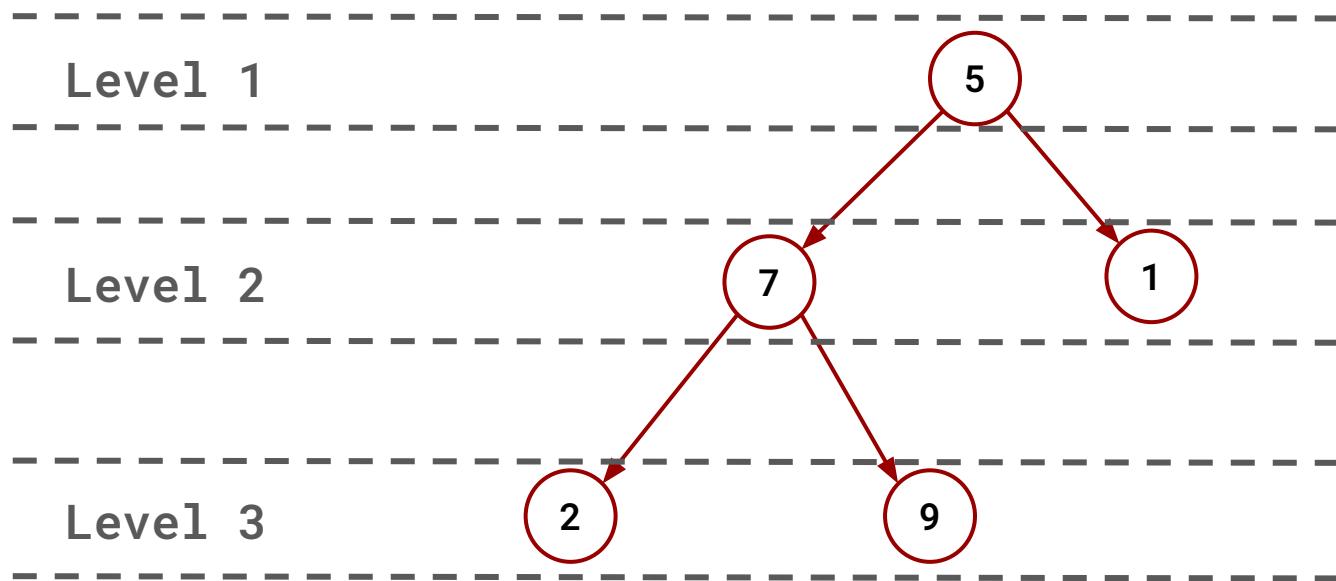


Δυαδικά Δέντρα (Binary Trees)



Επίπεδο Κόμβου (Node Level)

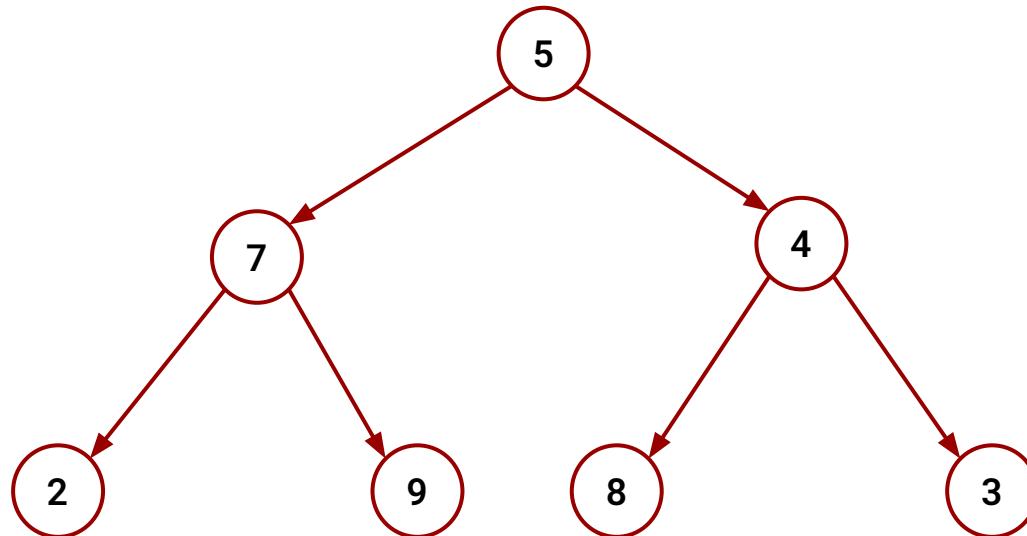
Το επίπεδο ενός κόμβου σε ένα δέντρο ισούται με τον αριθμό των κόμβων που μεσολαβούν μέχρι την ρίζα του δέντρου.



Τύποι Δυαδικών Δέντρων - Τέλειο (Perfect)

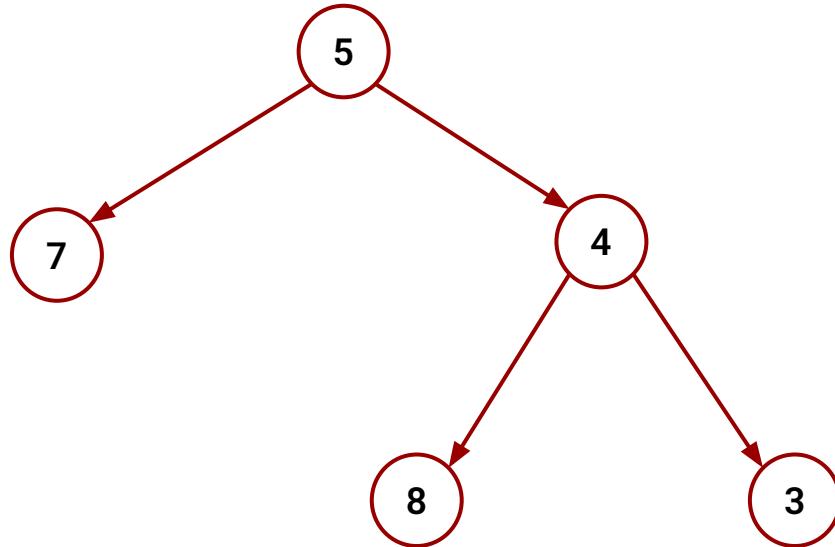
Τέλειο Δυαδικό Δέντρο (Perfect Binary Tree): ένα δυαδικό δέντρο που όλοι οι εσωτερικοί κόμβοι έχουν δύο παιδιά και όλα τα φύλλα βρίσκονται στο ίδιο επίπεδο.

Πόσοι κόμβοι σε η
επίπεδα;



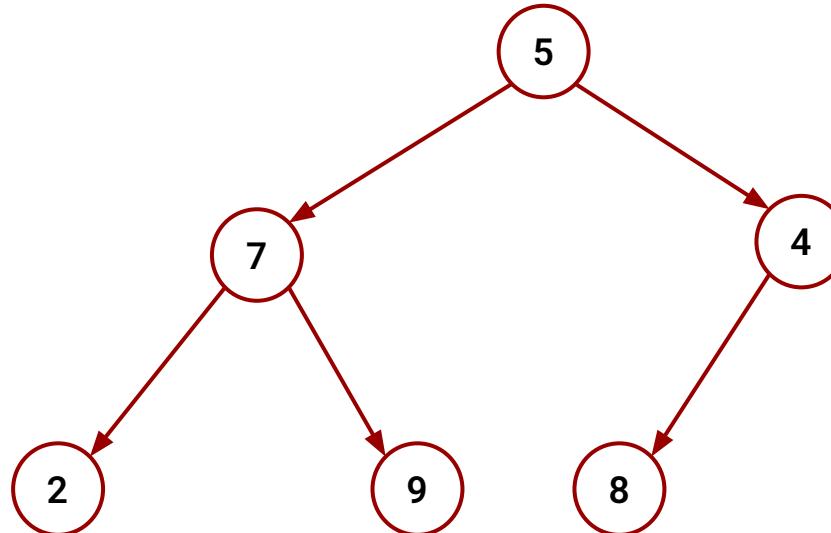
Τύποι Δυαδικών Δέντρων - Γεμάτο (Full)

Γεμάτο Δυαδικό Δέντρο (Full Binary Tree): ένα δυαδικό δέντρο που όλοι οι κόμβοι έχουν 0 ή 2 παιδιά.



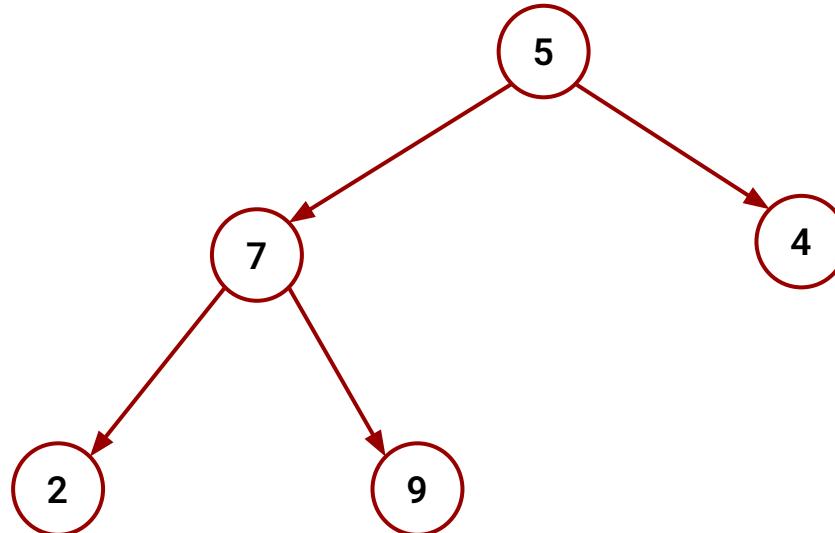
Τύποι Δυαδικών Δέντρων - Πλήρες (Complete)

Πλήρες Δυαδικό Δέντρο (Complete Binary Tree): ένα δυαδικό δέντρο που σε κάθε επίπεδο εκτός πιθανώς από το τελευταίο είναι γεμάτο και όλοι οι κόμβοι στο τελευταίο επίπεδο είναι όσο πιο αριστερά γίνεται.



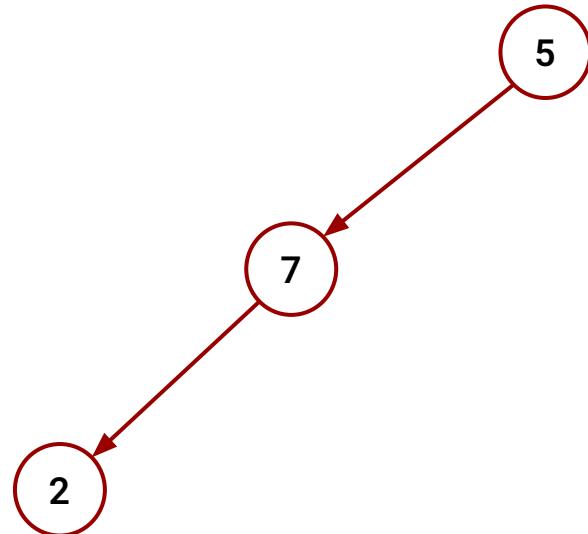
Τύποι Δυαδικών Δέντρων - Ισορροπημένο (Balanced)

Ισορροπημένο Δυαδικό Δέντρο (Balanced Binary Tree): ένα δυαδικό δέντρο που σε κάθε κόμβο το ύψος του αριστερού και το δεξιού υποδέντρου μπορούν να διαφέρουν μέχρι 1.



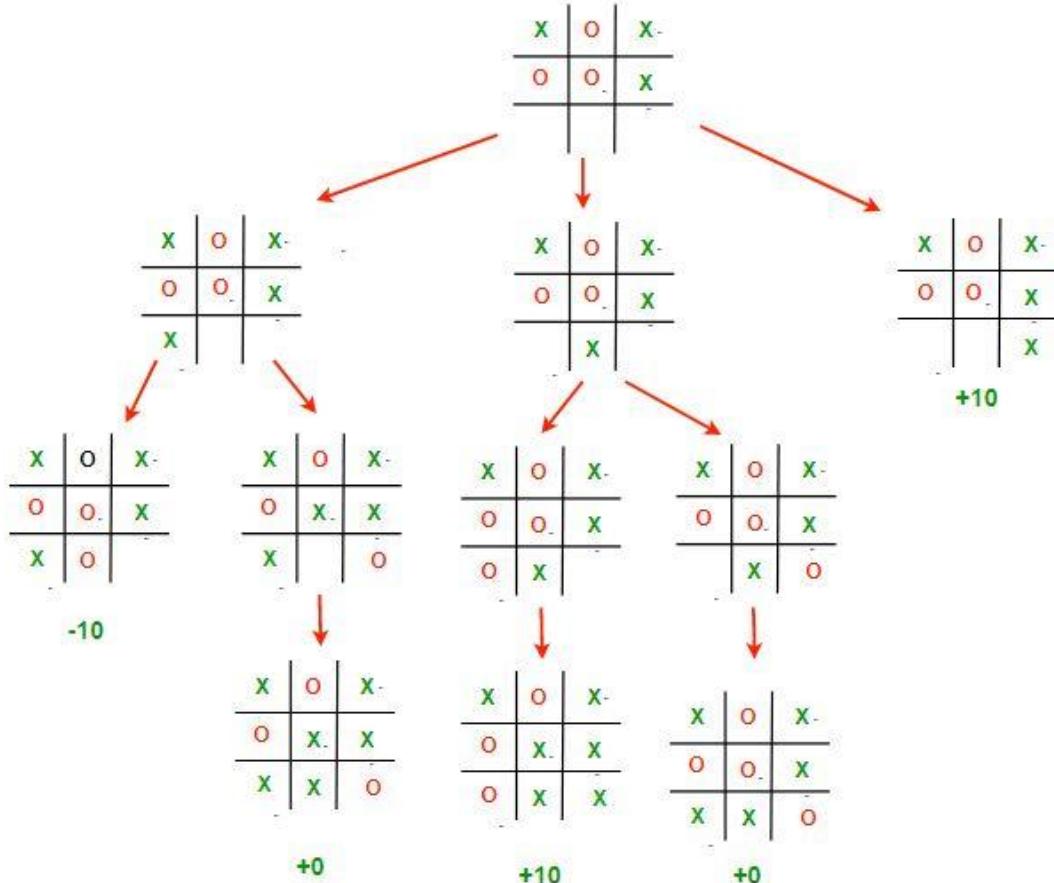
Τύποι Δυαδικών Δέντρων - Εκφυλισμένο (Degenerate)

Εκφυλισμένο Δυαδικό Δέντρο (Degenerate Binary Tree): ένα δυαδικό δέντρο που ο κάθε κόμβος έχει μέχρι ένα παιδί.



N-αδικά Δέντρα

Είναι συνηθισμένο να αναπαριστούμε καταστάσεις χρησιμοποιώντας δομές όπως τα δέντρα, κάποιες φορές με περισσότερα από 2 παιδιά κόμβους.



Βασικές Λειτουργίες Με Δυαδικά Δέντρα

1. is_empty: Έλεγχος εάν το δέντρο είναι άδειο
2. depth: Εύρεση βάθους δέντρου
3. print: Τύπωμα στοιχείων δέντρου
4. find: Εύρεση στοιχείου σε δέντρο
5. insert: Προσθήκη στοιχείου στο δέντρο (μόνοι σας)
6. delete: Αφαίρεση στοιχείου από δέντρο (μόνοι σας)

is_empty: Έλεγχος αν το δέντρο είναι άδειο

```
#include <stdio.h>
#include <stdlib.h>
typedef struct treenode {int value;  struct treenode * left;  struct treenode * right;} * Tree;
int is_empty(Tree t) {
    return t == NULL;
}
int main() {
    Tree t = NULL;
    printf("Empty: %d\n", is_empty(t));
    return 0;
}
```

\$./tree
Empty: 1

depth: Εύρεση βάθους δέντρου

```
#include <stdio.h>

#include <stdlib.h>

typedef struct treenode { int value; struct treenode * left; struct treenode * right;} * Tree;

int depth(Tree t) {

    if (t == NULL) return -1;

    int left_depth = depth(t->left);

    int right_depth = depth(t->right);

    return 1 + ((left_depth > right_depth) ? left_depth : right_depth);

}

int main() {

    struct treenode t2 = {2, NULL, NULL}, t9 = {9, NULL, NULL}, t1 = {1, NULL, NULL};

    struct treenode t7 = {7, &t2, &t9}; struct treenode t5 = {5, &t7, &t1};

    Tree t = &t5;

    printf("Depth: %d\n", depth(t));

    return 0;
}
```

depth: Εύρεση βάθους δέντρου

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

typedef struct treenode { int value; struct treenode * left; struct treenode * right;} * Tree;

int depth(Tree t) {
    if (t == NULL) return -1;

    int left_depth = depth(t->left);
    int right_depth = depth(t->right);

    return 1 + ((left_depth > right_depth) ? left_depth : right_depth);
}

int main() {
    struct treenode t2 = {2, NULL, NULL}, t9 = {9, NULL, NULL}, t1 = {1, NULL, NULL};

    struct treenode t7 = {7, &t2, &t9}; struct treenode t5 = {5, &t7, &t1};

    Tree t = &t5;
    printf("Depth: %d\n", depth(t));
    return 0;
}
```

```
$ ./depth
Depth: 2
```

Ποια η πολυπλοκότητα για ένα τέλειο δυαδικό δέντρο;

Χρόνος: $O(n)$
Χώρος: $O(\log n)$

Αναζήτηση κατά Βάθος (Depth-First Search or DFS)

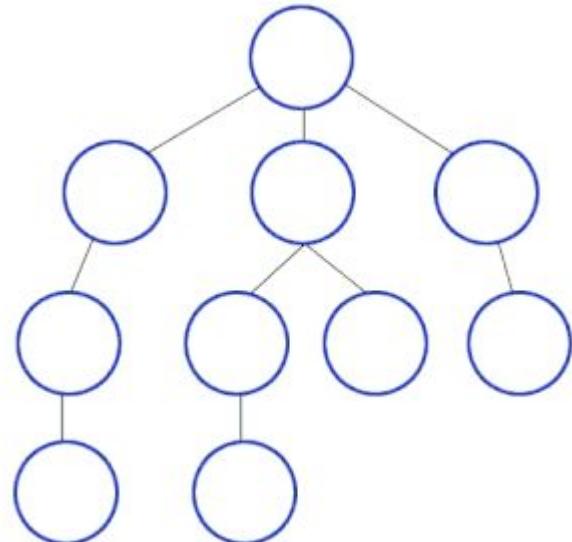
Η αναζήτηση κατά βάθος (depth-first search) είναι

ένας αλγόριθμος διάσχισης/αναζήτησης σε

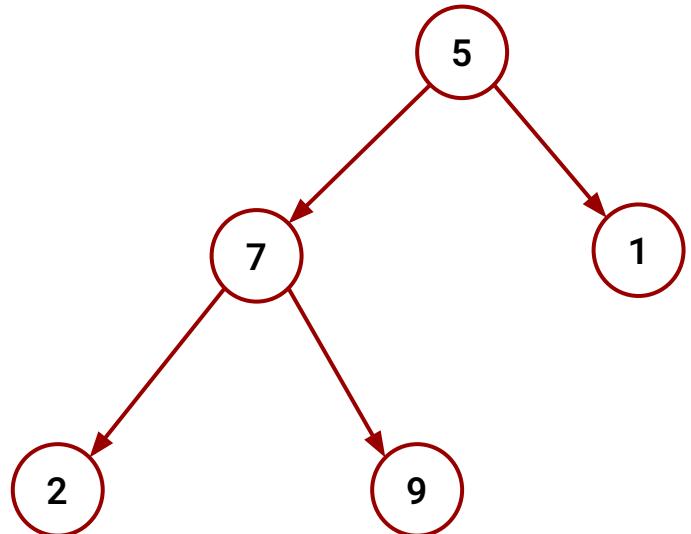
δέντρα και γράφους. Ο αλγόριθμος ξεκινά από τον

αρχικό κόμβο και εξερευνά όσο περισσότερο

μπορεί προτού οπισθοδρομίσει (backtracking).



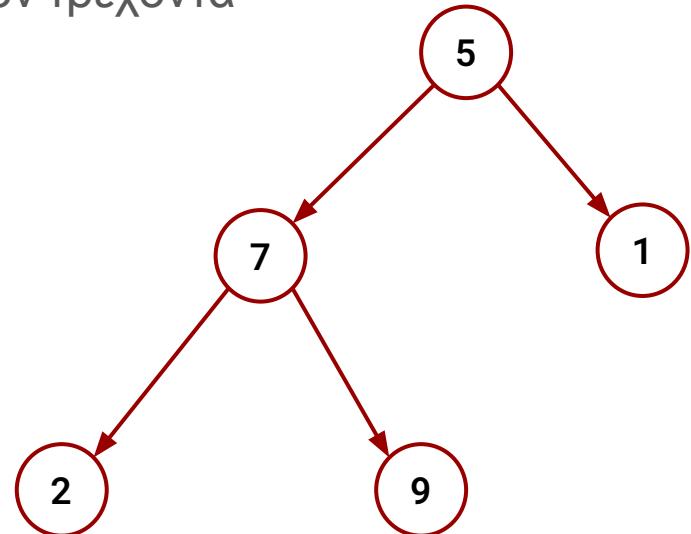
print: Τύπωμα & Διάσχιση (Traversal) Δέντρου



print: Τύπωμα & Διάσχιση (Traversal) Δέντρου

Pre-order Traversal: Τύπωσε (επεξεργάσου) πρώτα τον τρέχοντα κόμβο και μετά τα παιδιά

```
void print(Tree t) {  
    if (t == NULL) return;  
    printf("%d ", t->value);  
    print(t->left);  
    print(t->right);  
}
```

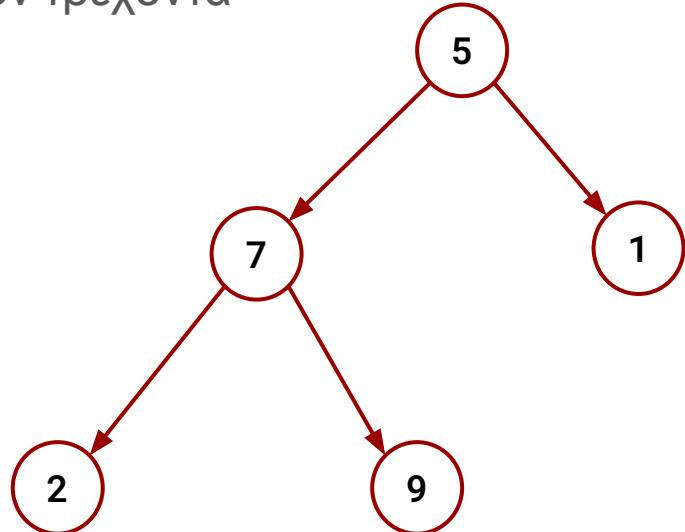


print: Τύπωμα & Διάσχιση (Traversal) Δέντρου

Pre-order Traversal: Τύπωσε (επεξεργάσου) πρώτα τον τρέχοντα κόμβο και μετά τα παιδιά

```
void print(Tree t) {  
    if (t == NULL) return;  
    printf("%d ", t->value);  
    print(t->left);  
    print(t->right);  
}
```

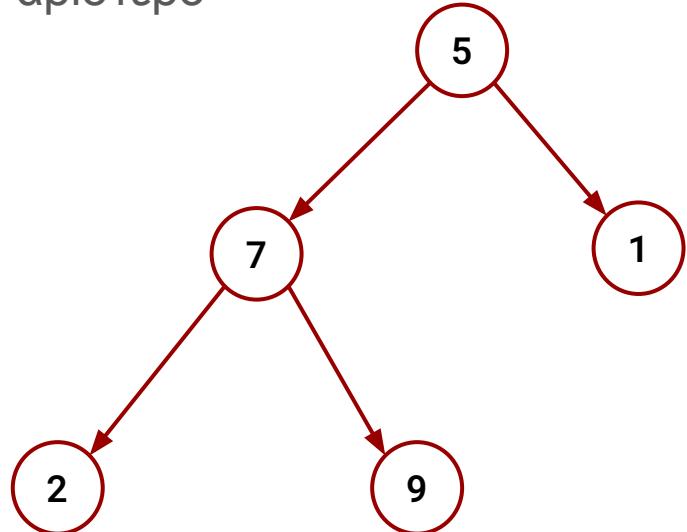
```
$ ./preorder  
5 7 2 9 1
```



print: Τύπωμα & Διάσχιση (Traversal) Δέντρου

In-order Traversal: Τύπωσε (επεξεργάσου) πρώτα τον αριστερό κόμβο, μετά τον τρέχοντα και τέλος τον δεξιό

```
void print(Tree t) {  
    if (t == NULL) return;  
    print(t->left);  
    printf("%d ", t->value);  
    print(t->right);  
}
```

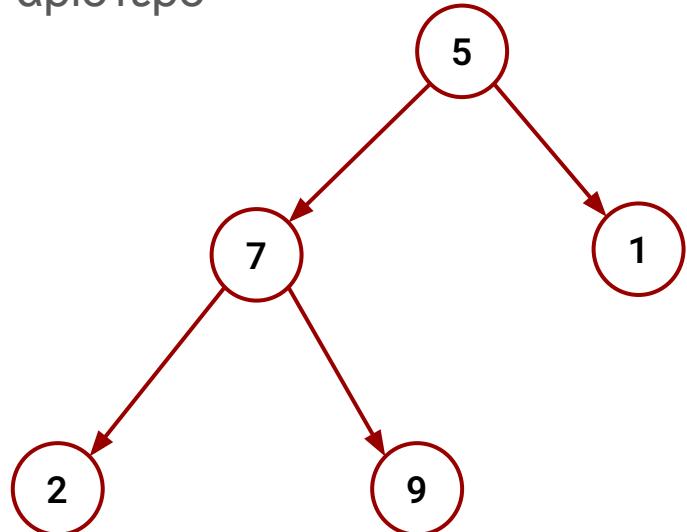


print: Τύπωμα & Διάσχιση (Traversal) Δέντρου

In-order Traversal: Τύπωσε (επεξεργάσου) πρώτα τον αριστερό κόμβο, μετά τον τρέχοντα και τέλος τον δεξιό

```
void print(Tree t) {  
    if (t == NULL) return;  
    print(t->left);  
    printf("%d ", t->value);  
    print(t->right);  
}
```

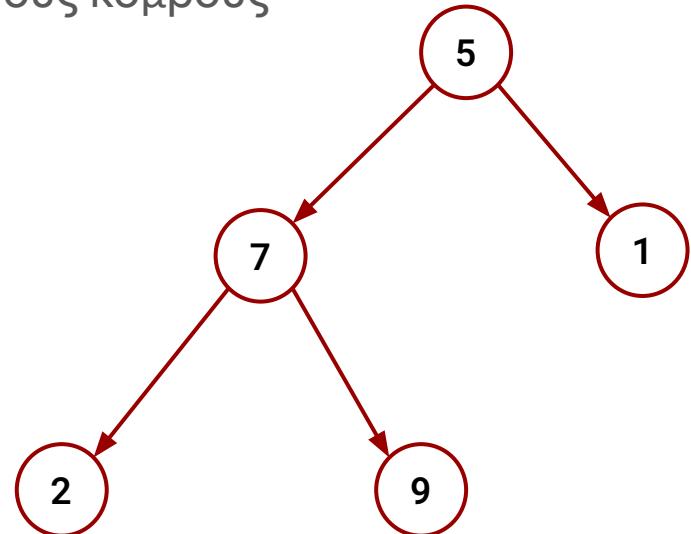
```
$ ./inorder  
2 7 9 5 1
```



print: Τύπωμα & Διάσχιση (Traversal) Δέντρου

Post-order Traversal: Τύπωσε (επεξεργάσου) πρώτα τους κόμβους παιδιά και στο τέλος τον τρέχοντα κόμβο

```
void print(Tree t) {  
    if (t == NULL) return;  
    print(t->left);  
    print(t->right);  
    printf("%d ", t->value);  
}
```

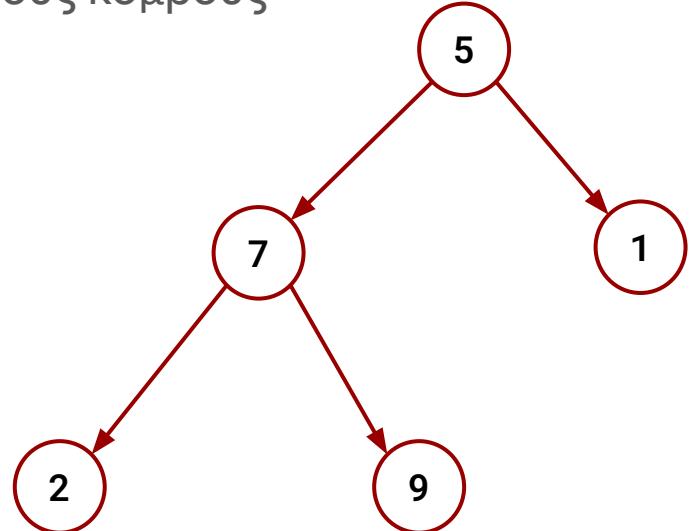


print: Τύπωμα & Διάσχιση (Traversal) Δέντρου

Post-order Traversal: Τύπωσε (επεξεργάσου) πρώτα τους κόμβους παιδιά και στο τέλος τον τρέχοντα κόμβο

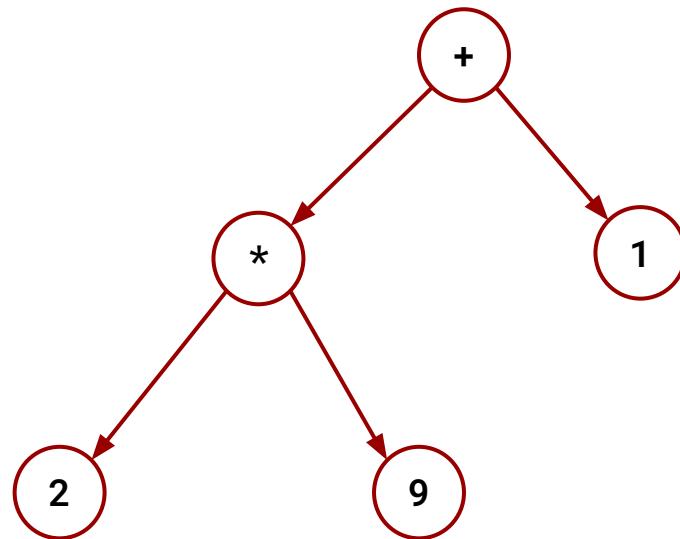
```
void print(Tree t) {  
    if (t == NULL) return;  
    print(t->left);  
    print(t->right);  
    printf("%d ", t->value);  
}
```

```
$ ./postorder  
2 9 7 1 5
```



Χρόνος: $O(n)$
Χώρος: $O(\log n)$

Θέλω να γράψω έναν αποτιμητή εκφράσεων (calculator / evaluator / interpreter). Ποια διάσχιση θα χρησιμοποιήσω;



find: Αναζήτηση Στοιχείου σε Δυαδικό Δέντρο

```
Tree find(Tree t, int value) {  
    if (t == NULL) return NULL;  
    if (t->value == value) return t;  
    Tree left = find(t->left, value);  
    if (left != NULL) return left;  
    return find(t->right, value);  
}
```

Χρόνος: $O(n)$
Χώρος: $O(\log n)$

Αναζήτηση κατά Πλάτος (Breadth-First Search or BFS)

Η αναζήτηση κατά πλάτος ή κατά επίπεδα

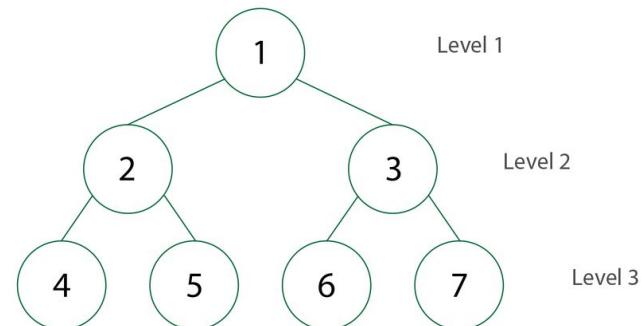
(breadth-first search - BFS) είναι ένας αλγόριθμος

διάσχισης/αναζήτησης σε δέντρα και γράφους. Ο

αλγόριθμος ξεκινά από τον αρχικό κόμβο και σε κάθε

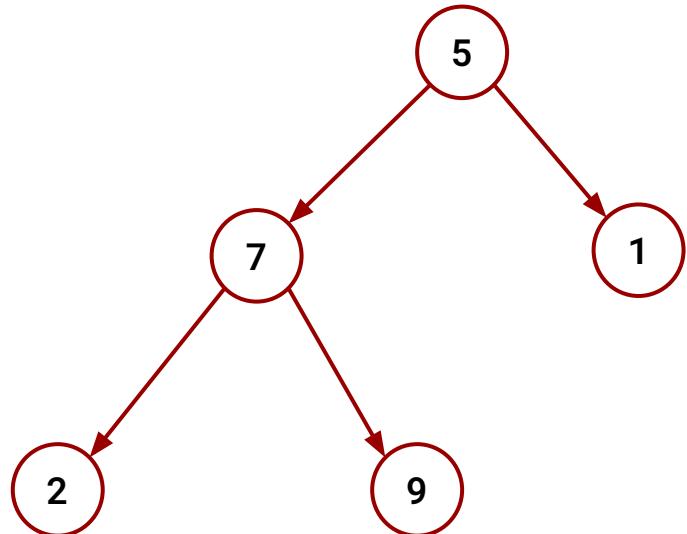
βήμα εξερευνά όλους τους κόμβους στο τρέχον

επίπεδο προτού περάσει στο επόμενο.



Τύπωμα με Διάσχιση κατά Πλάτος - BFS

```
void bfs(Tree t) {  
    List worklist = NULL;  
    Tree tmp;  
    insert(&worklist, t);  
    while(worklist) {  
        tmp = pop_last(&worklist);  
        printf("%d ", tmp->value);  
        if (tmp->left) insert(&worklist, tmp->left);  
        if (tmp->right) insert(&worklist, tmp->right);  
    }  
}
```



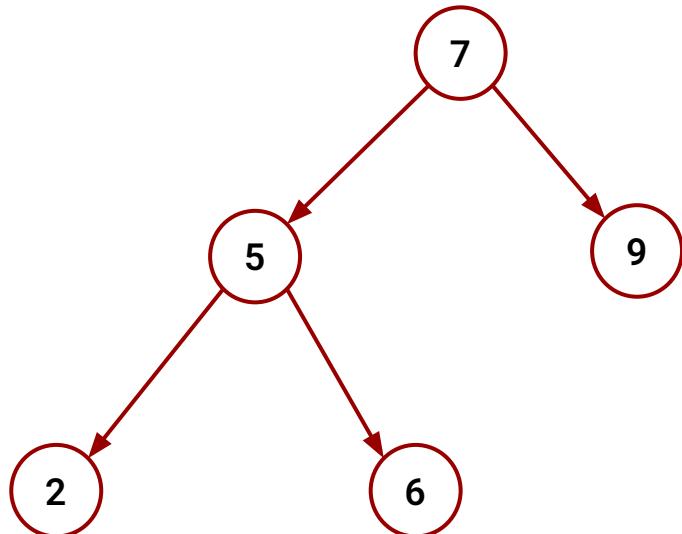
Τύπωμα με Διάσχιση κατά Πλάτος - BFS

```
void bfs(Tree t) {  
    List frontier = NULL;  
    Tree tmp;  
    insert(&frontier, t);  
    while(frontier) {  
        tmp = pop_last(&frontier);  
        printf("%d ", tmp->value);  
        if (tmp->left) insert(&frontier, tmp->left);  
        if (tmp->right) insert(&frontier, tmp->right);  
    }  
}
```

Χρόνος: $O(n)$
Χώρος: $O(n)$

Δυαδικό Δέντρο Αναζήτησης (Binary Search Tree - BST)

Ένα **Δυαδικό Δέντρο Αναζήτησης** (Binary Search Tree - BST ή Ordered Tree) είναι ένα ταξινομημένο δέντρο έτσι ώστε κάθε αριστερός κόμβος να είναι μικρότερος του γονιού του και κάθε δεξιός κόμβος να είναι μεγαλύτερος του.



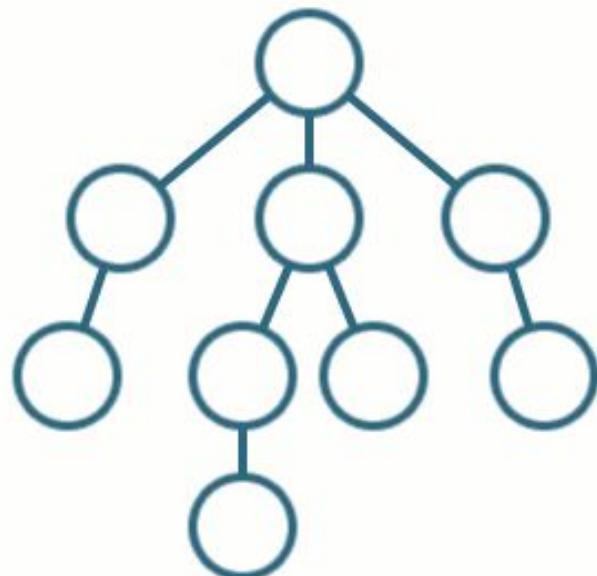
Έλεγχος αν υπάρχει (exists) ένα στοιχείο σε δυαδικό δέντρο. Πως;

```
int exists(Tree t, int value) {  
    if (t == NULL) return 0;  
    if (t->value == value) return 1;  
    if (value < t->value) return exists(t->left, value);  
    return exists(t->right, value);  
}
```

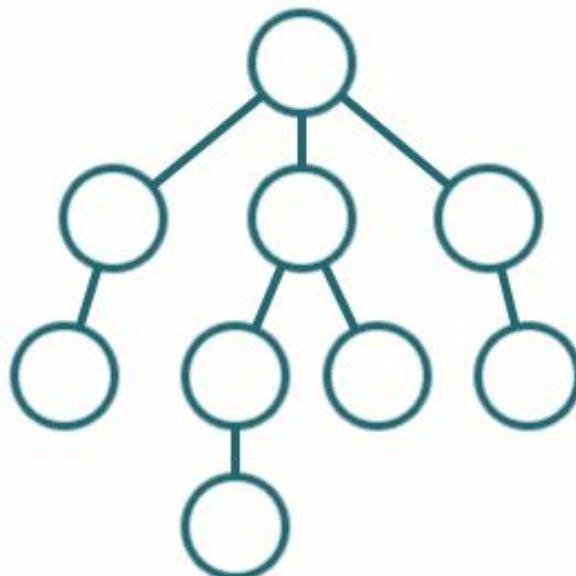
Χρόνος: $O(\log n)$
Χώρος: $O(1)$

Ποιος αλγόριθμος αναζήτησης είναι καλύτερος;

DFS



BFS



Έχεις να αποθηκεύσεις 1 PetaByte (10^6 GB) δεδομένων για μια υπηρεσία (service). Πως θα αποθηκεύσεις τα δεδομένα σου; Πως θα κάνεις αναζήτηση;

Γράφεις ένα πρόγραμμα που βρίσκει λύσεις σε δέντρα-λαβυρίνθους. Αναζητάς το συντομότερο μονοπάτι για την έξοδο. BFS ή DFS; Γιατί;

Θέλεις να βρεις το μέγιστο στοιχείο ενός δέντρου. Προτιμάς BFS ή DFS; Γιατί;

Διάλεξη 22 - Οργάνωση Κώδικα

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

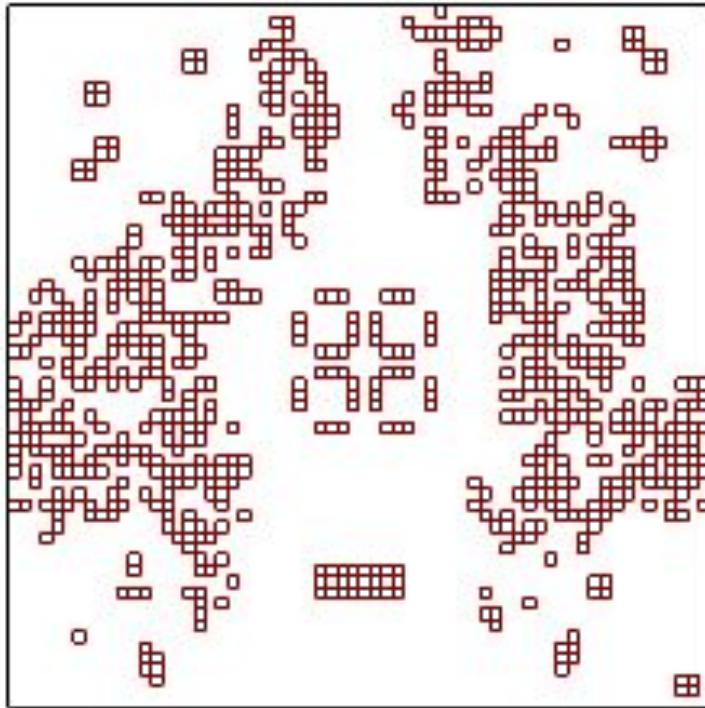
Θανάσης Αυγερινός

Γραμμές Κώδικα (Lines of Code - LOC)

Η γραμμή κώδικα (Line of Code/Source Line of Code - LOC/SLOC), δηλαδή οι εντολές που γράφουμε μέχρι την αλλαγή γραμμής (newline) είναι μία από τις βασικές μετρικές για να κατανοήσουμε το μέγεθος προγραμμάτων.

- LOC
- KLOC = 10^3 LOC
- MLOC = 10^6 LOC
- ...

Με λίγες γραμμές κώδικα, μπορούμε να πάρουμε
ιδιαίτερα σύνθετα συστήματα



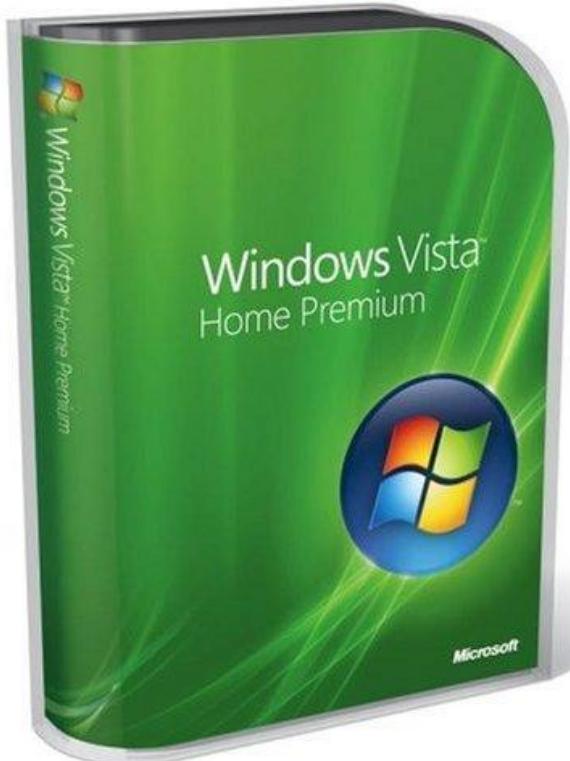
Conway's Game of Life (1970)

~100-200 LOC

Apollo 11 (1969)

145 KLOC





Windows Vista (2006)

50 MLOC

Πόσες γραμμές γράψαμε στις εργασίες μας;



```
$ sloccount hw-submissions/
```

```
Total Physical Source Lines of Code (SLOC) = 67,826
```

```
Development Effort Estimate, Person-Years (Person-Months) = 16.75 (200.99)
```

```
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
```

```
Schedule Estimate, Years (Months) = 1.56 (18.76)
```

```
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
```

```
Estimated Average Number of Developers (Effort/Schedule) = 10.72
```

```
Total Estimated Cost to Develop = $ 2,262,599
```

```
(average salary = $56,286/year, overhead = 2.40).
```

```
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
```

```
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
```

Υλοποιούμε ένα καινούριο σύστημα και περιμένουμε να έχει ~40 χιλιάδες γραμμές κώδικα. Τι κάνουμε;

Λύση #1: Όλος ο κώδικας σε ένα αρχείο C

Θετικά

1. Απλή οργάνωση, εύκολη μεταφορά, όλος ο κώδικας σε ένα μέρος
2. Οι ορισμοί όλων των συναρτήσεων προσβάσιμοι στο ίδιο αρχείο

Αρνητικά

1. Το να ψάχνεις να βρεις κάτι σε ένα αρχείο με δεκάδες χιλιάδες γραμμές είναι οδυνηρό
2. Αλλάζεις μια γραμμή κώδικα και πρέπει να κάνεις compile τα πάντα
3. Συντήρηση, αναβάθμιση, κατανόηση όλου του προγράμματος δύσκολη

Iδέα: Abstraction (αφαίρεση;) και διάσπαση σε
υποπροβλήματα

Λύση #2: Οργάνωση του κώδικα σε πολλά αρχεία

Κάθε αρχείο περιέχει μεταβλητές και συναρτήσεις που σχετίζονται θεματικά, λειτουργικά ή σύμφωνα με άλλα κριτήρια, π.χ. [openssl](#):

```
├── README.md  
├── ssl  
|   ├── ssl_init.c  
|   ├── event_queue.c  
|   ├── ssl_err.c  
|   ├── sslerr.h  
|   ...  
└── test  
    ├── aborttest.c  
    └── acvp_test.c  
...
```

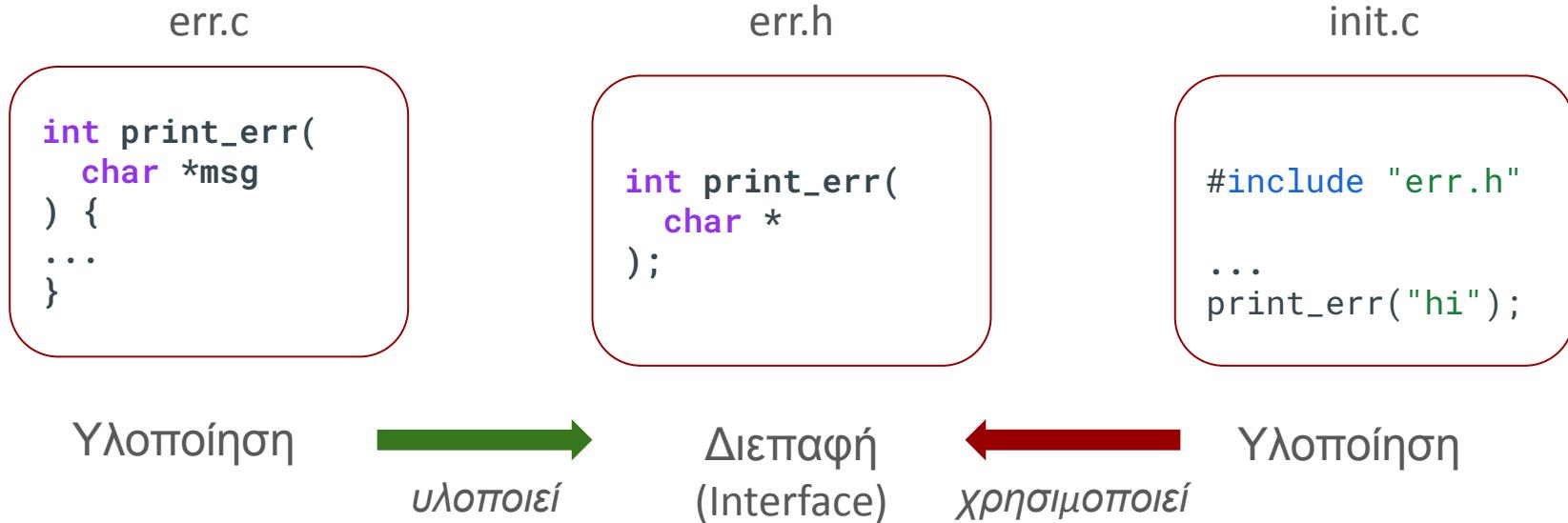
Αρχεία Υλοποίησης (.c)

Αρχεία Κεφαλίδας (.h)

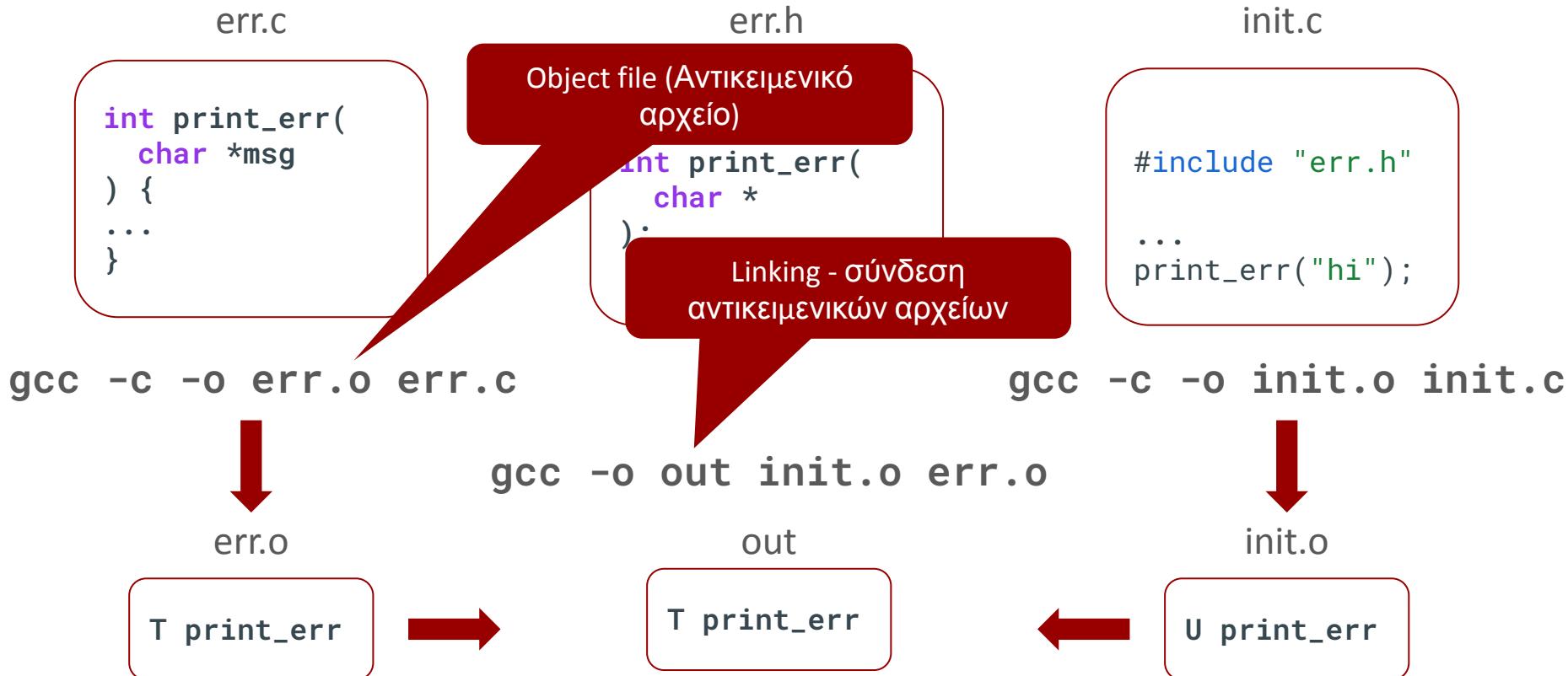
Ποιος είναι ο καλύτερος τρόπος να οργανώσουμε τον κώδικα μας;

Εξαρτήσεις (Dependencies)

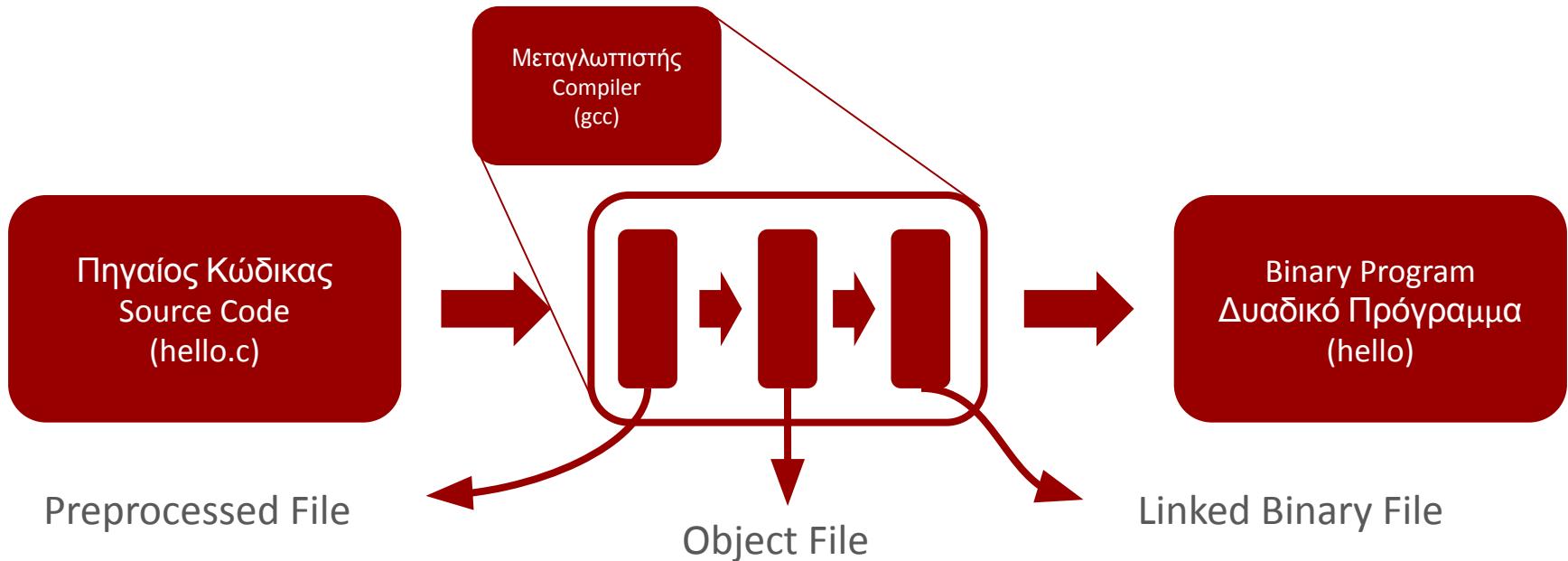
Αν αλλάξω ένα αρχείο τι επηρεάζεται;
Με τι μοιάζουν αυτές οι εξαρτήσεις;



Μεταγλώττιση Με Πολλά Αρχεία



Μεταγλώττιση

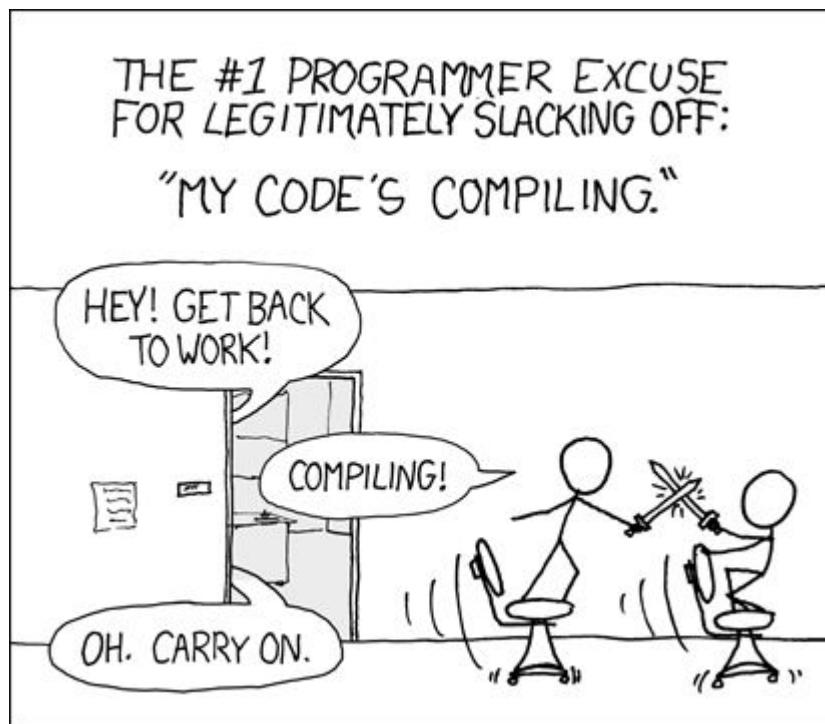


Η Μεταγλώττιση είναι Χρονοβόρα Διαδικασία

Σε μεγάλα project όπως ο πυρήνας του Linux η μεταγλώττιση μπορεί να πάρει **ώρες** (ή ακόμα και μέρες!)

Σπάζοντας το πρόγραμμα σε αρχεία μπορούμε να γλυτώσουμε χρόνο μεταγλωττίζοντας μόνο ότι χρειάζεται μετά από κάθε αλλαγή ([Makefiles](#))

Μετά την πρώτη μεταγλώττιση, οι επόμενες επαναλήψεις είναι συνήθως γρηγορότερες



Η Μεταγλώττιση είναι γραμμική διαδικασία (στην C)

```
#include <stdio.h>

int main() {
    print_err("hello");
    return 0;
}

int print_err(char * msg) {
    return fprintf(stderr, "%s\n", msg);
}
```

```
$ gcc -o prototype prototype.c
prototype.c: In function 'main':
prototype.c:4:3: warning: implicit declaration
of function 'print_err'
[-Wimplicit-function-declaration]
4 |     print_err("hello");
|     ^~~~~~
```

Δήλωση Πρωτοτύπου Συνάρτησης (Function Prototype)

Η δήλωση του πρωτοτύπου μιας συνάρτησης (function prototype) καθορίζει το όνομα της συνάρτησης, τον τύπο επιστροφής της και τα ορίσματά της.

τύπος όνομα(λίστα_ορισμάτων);

Για παράδειγμα:

```
int print_err(char * message);
```

```
int print_err(char *);
```

Τα ονόματα των ορισμάτων
μπορούν να παραληφθούν

Η Μεταγλώττιση είναι γραμμική διαδικασία (στην C)

```
#include <stdio.h>
int print_err(char *msg);
int main() {
    print_err("hello");
    return 0;
}
int print_err(char * msg) {
    return fprintf(stderr, "%s\n", msg);
}
```

Προσθήκη Πρωτότυπου

```
$ gcc -o prototype prototype.c
$ ./prototype
hello
```