

Διάλεξη 23 - Οργάνωση Κώδικα

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός / Τάκης Σταματόπουλος

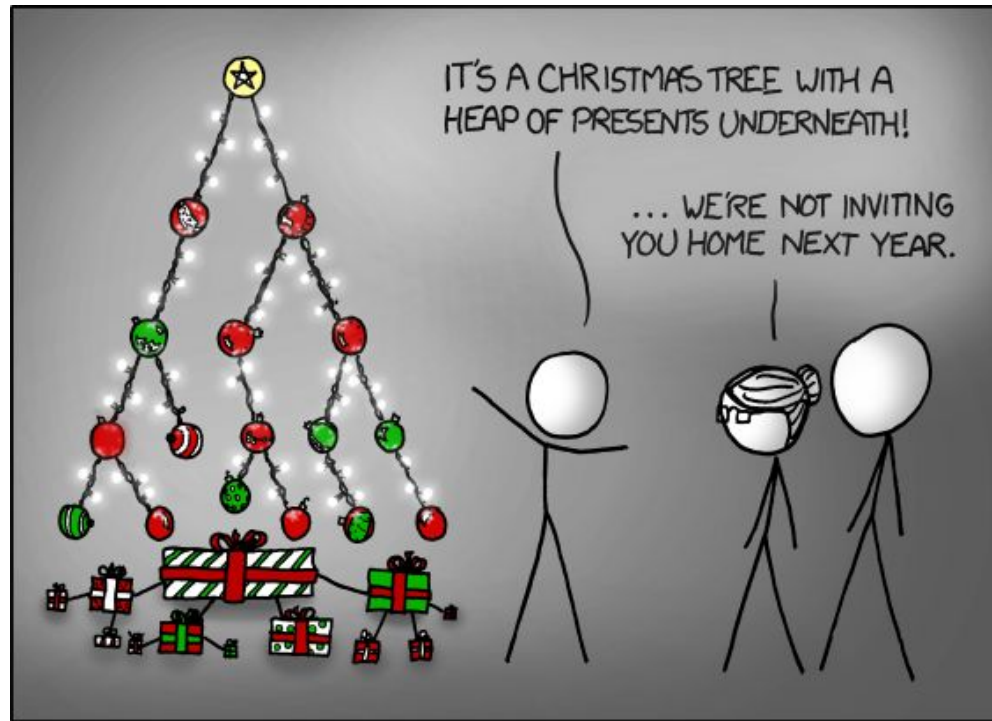
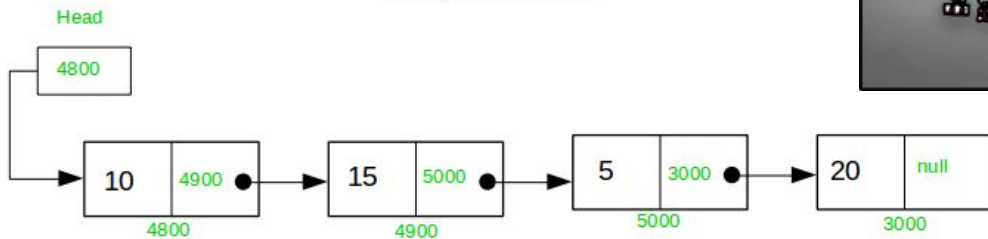
Ανακοινώσεις / Διευκρινίσεις

- Βγήκε η Εργασία #2 - Προθεσμία: 23:59, Σάββατο 10 Ιανουαρίου 2026

Την Προηγούμενη Φορά

- Αυτοαναφορικές Δομές
 - Λίστες
 - Δέντρα
 - Αλγόριθμοι χρήσης και διάσχισης

Singly Linked list



Σήμερα

- Μεγάλα Προγράμματα, Οργάνωση και Μεταγλώττιση
- Δηλώσεις μεταβλητών και συναρτήσεων
- Const
- Makefiles - Guest lecture με τον βοηθό του μαθήματος Πέτρο Πλαστήρα

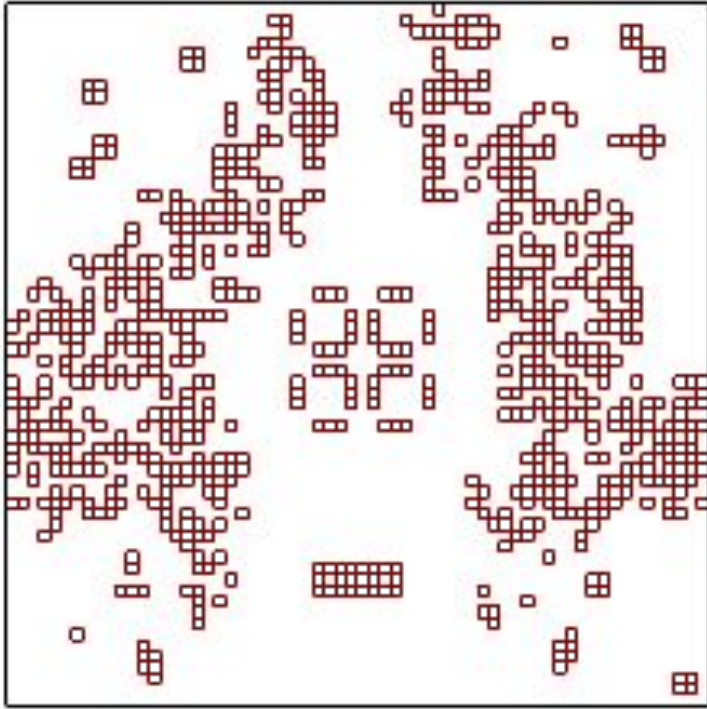


Γραμμές Κώδικα (Lines of Code - LOC)

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
ιδιαιτέρα σύνθετα συστήματα



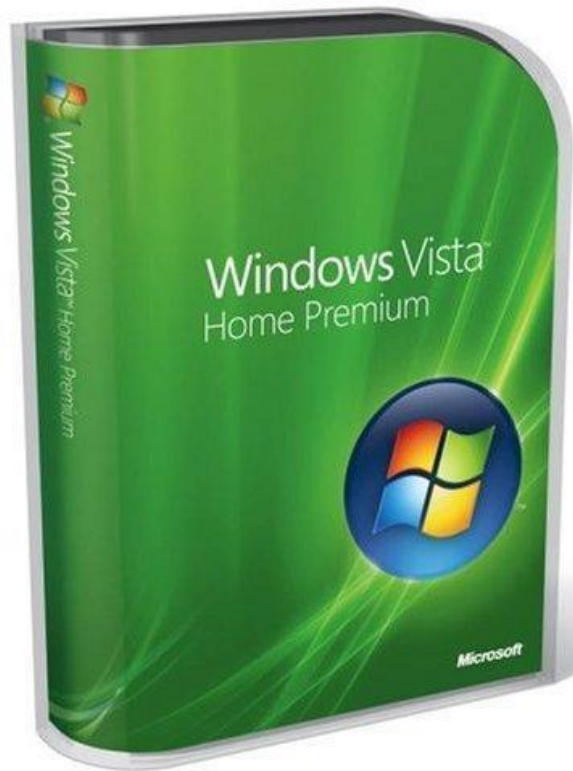
Conway's Game of Life (1970)

~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

Ιδέα: 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 κώδικά μας;

Εξαρτήσεις (Dependencies)

Αν αλλάξω ένα αρχείο τι επηρεάζεται;
Με τι μοιάζουν αυτές οι εξαρτήσεις;

err.c

```
int print_err(  
    char *msg  
) {  
    ...  
}
```

Υλοποίηση



υλοποιεί

err.h

```
int print_err(  
    char *  
);
```

Διεπαφή
(Interface)



χρησιμοποιεί

init.c

```
#include "err.h"  
  
...  
print_err("hi");
```

Υλοποίηση

Μεταγλώττιση Με Πολλά Αρχεία

err.c

```
int print_err(  
    char *msg  
) {  
    ...  
}
```

err.h

```
int print_err(  
    char *  
);
```

init.c

```
#include "err.h"  
  
...  
print_err("hi");
```

Object file (Αντικειμενικό
αρχείο)

Linking - σύνδεση
αντικειμενικών αρχείων

gcc -c -o err.o err.c

err.o

T print_err

gcc -c -o init.o init.c

init.o

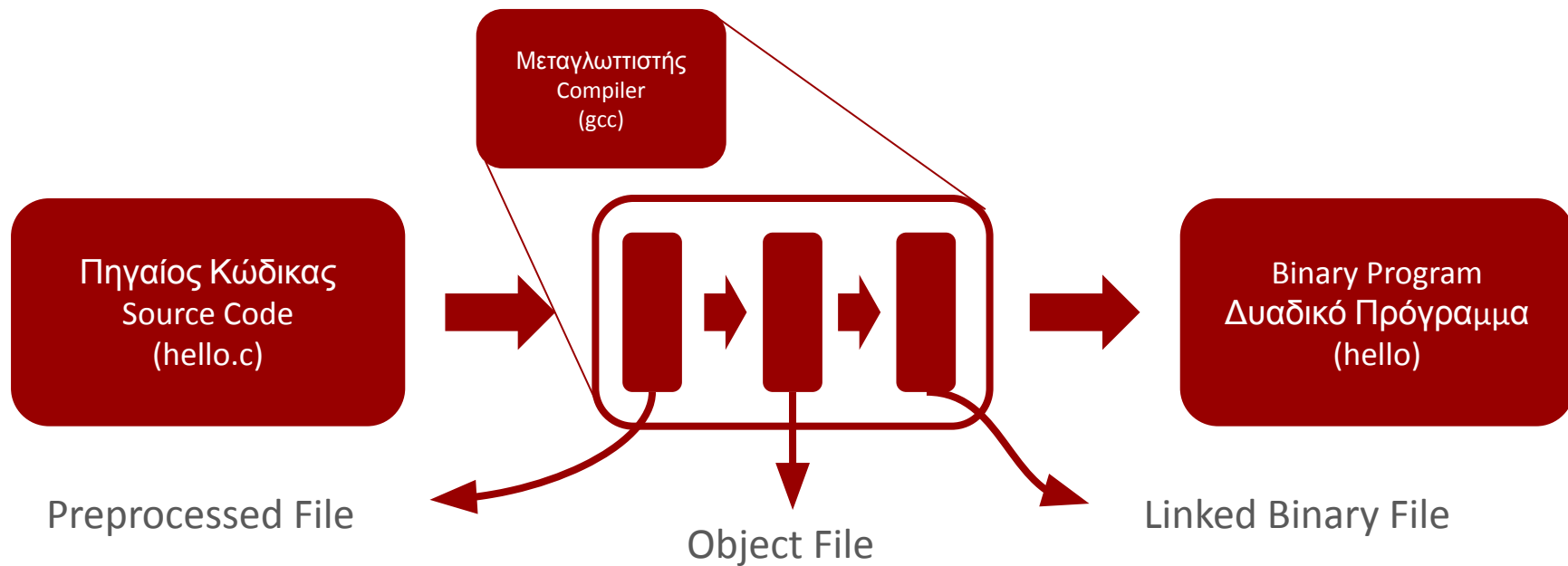
U print_err

gcc -o out init.o err.o

out

T print_err

Μεταγλώττιση

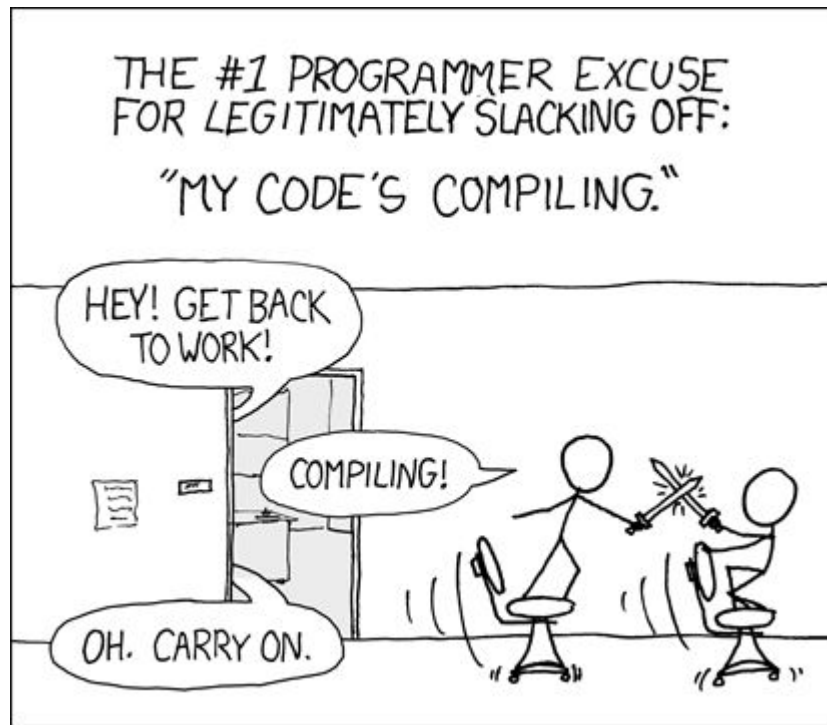


Η Μεταγλώττιση είναι Χρονοβόρα Διαδικασία

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 παραλειφθούν

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 Πρωτοτύπου

```
$ gcc -o prototype prototype.c
```

```
$ ./prototype
```

```
hello
```

Δήλωση μεταβλητών ως σταθερών (const)

Ο προσδιοριστής **const (const qualifier)** μπορεί να χρησιμοποιηθεί για να ορίσει ότι το περιεχόμενο κάποιων θέσεων μνήμης είναι σταθερό. Γενική σύνταξη:

```
const δήλωση_μεταβλητής;
```

Παραδείγματα:

```
const int x = 42;
```

```
const char message[] = "Hello";
```

```
const char const * msg_ptr = message;
```

Αν κάτι δηλωθεί ως `const` *δεν* πρέπει να το αλλάξουμε

```
const char message[] = "Hello";
```

```
int main() {
```

```
    const int x = 42;
```

```
    const char const * msg_ptr = message;
```

```
    x++;
```

```
    return 0;
```

```
}
```

```
$ gcc -o const1 const1.c
const1.c: In function 'main':
const1.c:5:4: error: increment
of read-only variable 'x'
```

```
5 | x++;
  | ^~
```

Δεν αλλάζουμε const θέσεις μνήμης

```
const char message[] = "Hello";  
  
int main() {  
    const int x = 42;  
  
    char * bad = (char*)message;  
  
    bad[1] = 'o';  
  
    return 0;  
}
```

```
$ gcc -o const2 const2.c  
$ ./const2  
Segmentation fault
```


Χρήση const σε ορισμούς συναρτήσεων

Χρησιμοποιώντας const στις δηλώσεις συναρτήσεων (ή πρωτοτύπων) δημιουργούμε **συμβόλαια (contracts)** με τους χρήστες της συνάρτησής μας

Για παράδειγμα:

```
int print_err(const char * message);
```

Με το παραπάνω εγγυόμαστε ότι η `print_err` δεν θα αλλάξει τους χαρακτήρες της `message`.

Τέτοιες εγγυήσεις επιτρέπουν και στους προγραμματιστές να είναι πιο αποδοτικοί αλλά και στους μεταγλωττιστές να γράφουν πιο γρήγορο κώδικα.

Αναφορές

Από τις διαφάνειες του κ. Σταματόπουλου έχουμε πλέον καλύψει όλη την ύλη

- [Lines of Code Written](#)
- [Linux Kernel surpasses 40MLOC](#)

Ευχαριστώ και καλή μέρα εύχομαι!
Keep Coding ;)