

Διάλεξη 24 - Προχωρημένα Θέματα

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

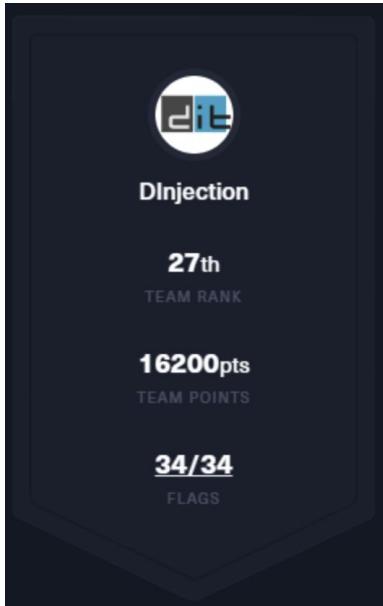
Θανάσης Αυγερινός / Τάκης Σταματόπουλος

Ανακοινώσεις / Διευκρινίσεις

- Η Εργασία #3 θα βγει μέσα στην βδομάδα. Go Project μέχρι 2 άτομα. Προθεσμίες:
17 Ιανουαρίου για βασική υποβολή / τέλος εξαμήνου για διαγωνισμό.



UniCTF 2025 Results - Hack-The-Box (2025/12/19)

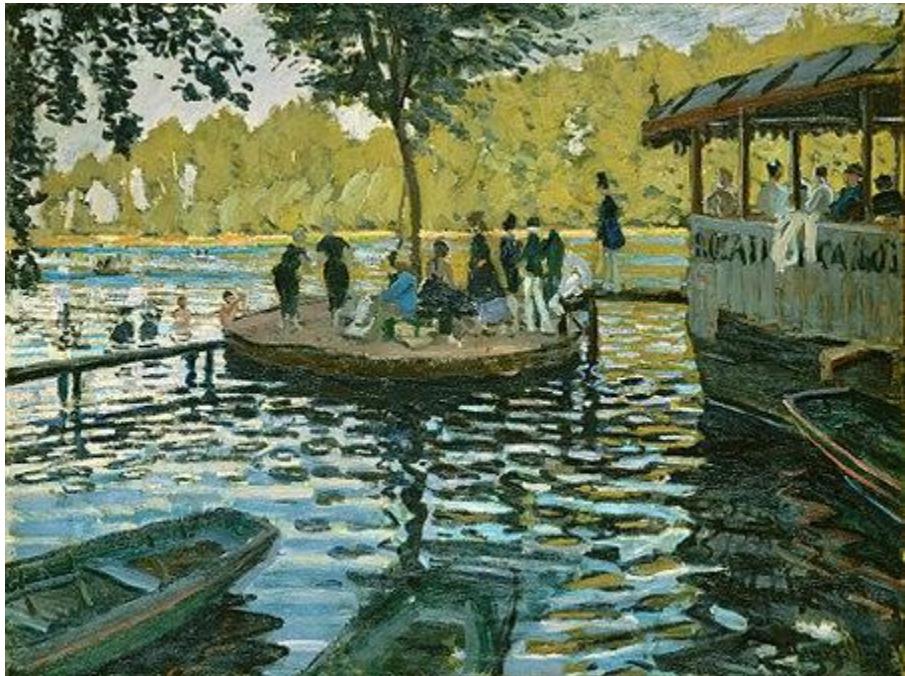


| | | | | |
|----|-----------------------------------|----|-------|-------|
| 27 | DInjection | GR | 16200 | 34/34 |
| 28 | Technical Cluj-Napoca | RO | 15250 | 33/34 |
| 29 | Universidad del Gran Rosario | AR | 15250 | 33/34 |
| 30 | University of East London | GB | 15250 | 33/34 |
| 31 | Universitat Autònoma de Barcelona | ES | 15250 | 33/34 |
| 32 | Abu Dhabi University | AE | 15250 | 33/34 |
| 33 | GIU | EG | 15250 | 33/34 |
| 34 | UoEsglnt | GB | 15225 | 33/34 |
| 35 | NTUA | GR | 15225 | 33/34 |

27η θέση στον κόσμο, 1η θέση στην Ελλάδα (μοναδική που έλυσε όλα τα προβλήματα), συμμετείχαν και πρωτοετείς!?

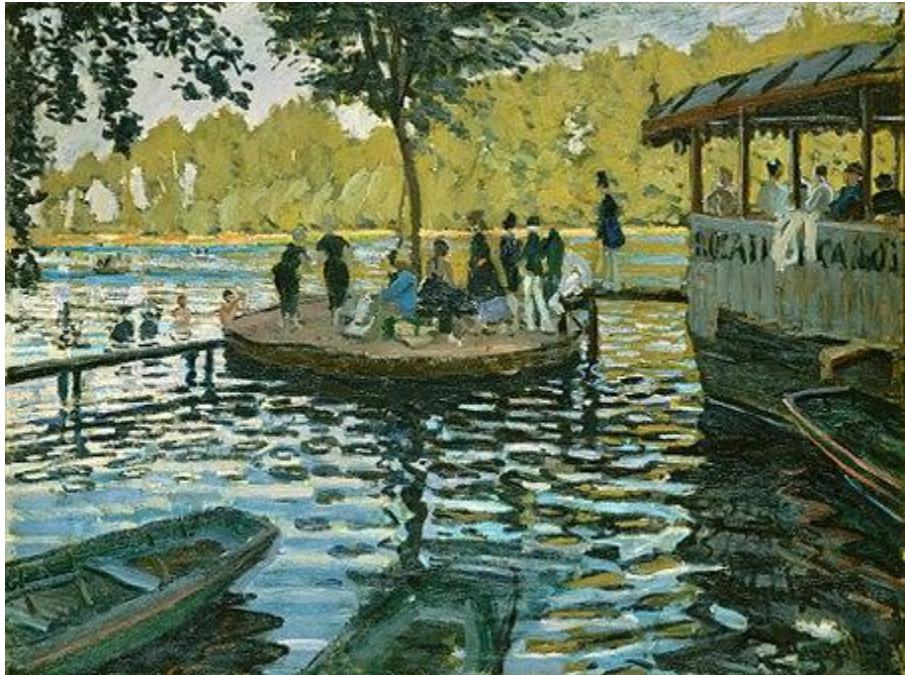
Την Προηγούμενη Φορά

- Μεγάλα Προγράμματα, Οργάνωση και Μεταγλώττιση
- Δηλώσεις μεταβλητών και συναρτήσεων
- const



Σήμερα

- Μεγάλα Προγράμματα, Οργάνωση και Μεταγλώττιση
- Δηλώσεις μεταβλητών και συναρτήσεων
- const
- Και άλλα προχωρημένα θέματα

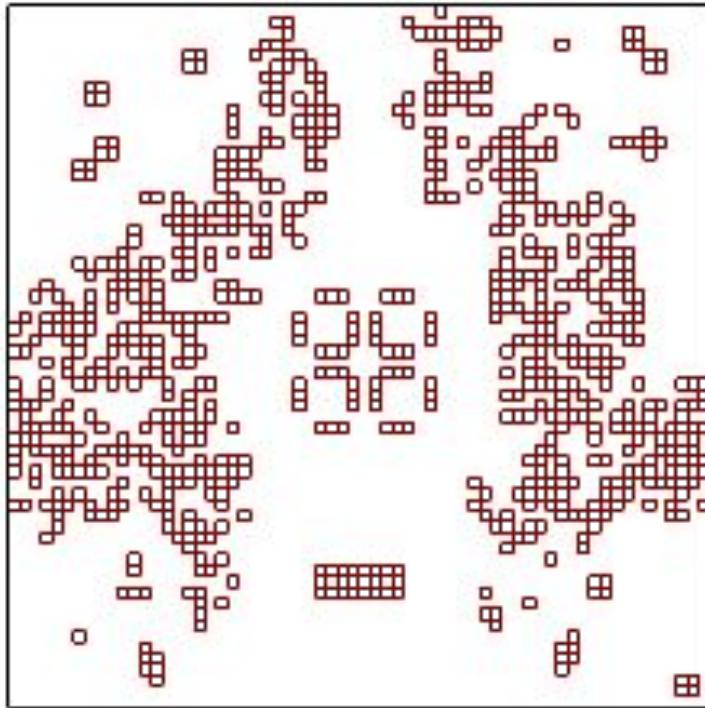


Γραμμές Κώδικα (Lines of Code - LOC)

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



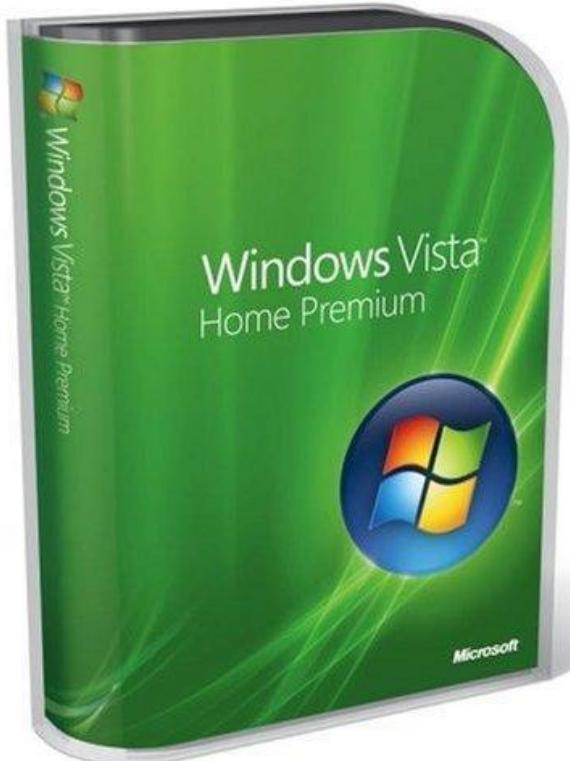
Conway's Game of Life (1970)

~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) = 119,059
```

```
Development Effort Estimate, Person-Years (Person-Months) = 30.24 (362.88)
```

```
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
```

```
Schedule Estimate, Years (Months) = 1.96 (23.48)
```

```
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
```

```
Estimated Average Number of Developers (Effort/Schedule) = 15.46
```

```
Total Estimated Cost to Develop = $ 4,084,999
```

```
(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  
...
```

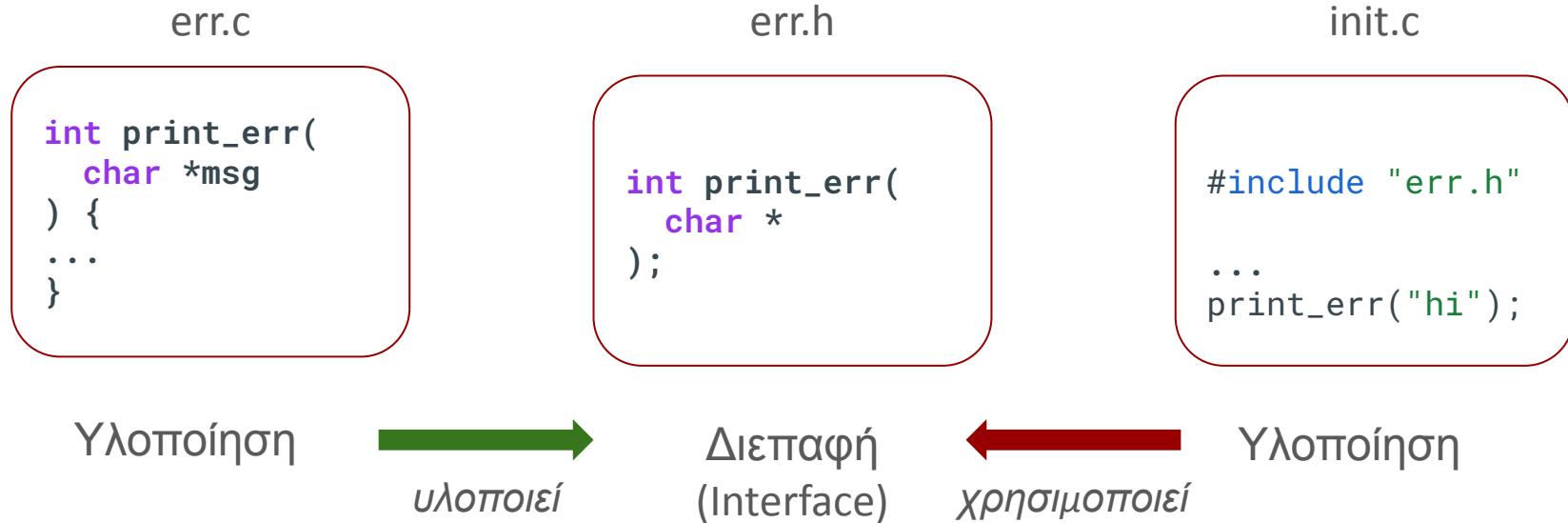
Αρχεία Υλοποίησης (.c)

Αρχεία Κεφαλίδας (.h)

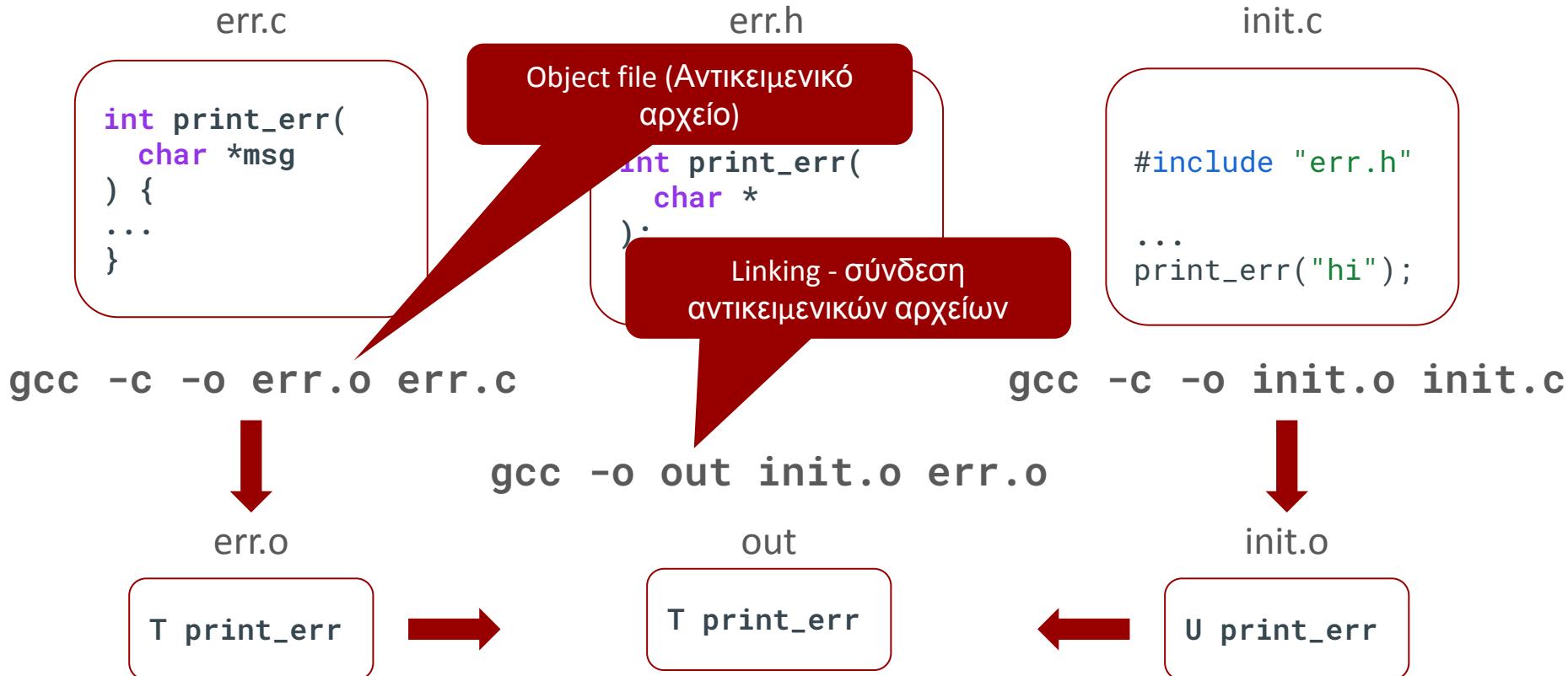
Ποιος είναι ο καλύτερος τρόπος να οργανώσουμε τον κώδικα μας;

Εξαρτήσεις (Dependencies)

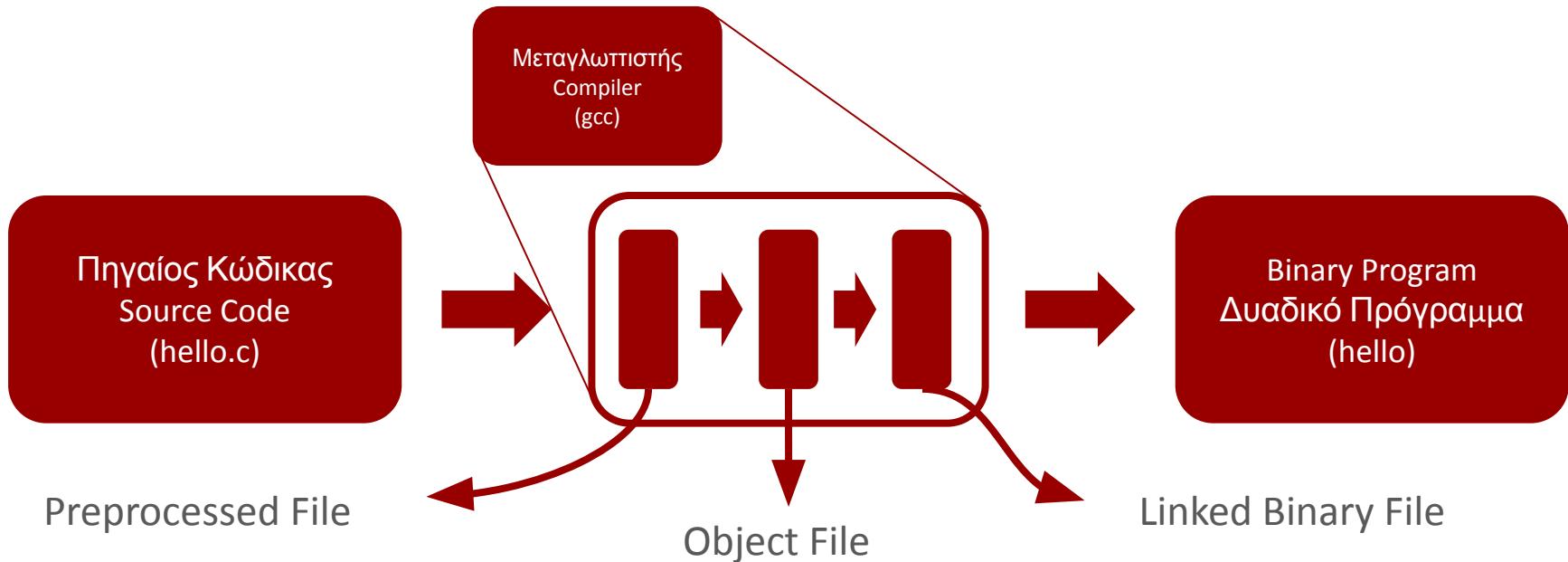
Αν αλλάξω ένα αρχείο τι επηρεάζεται;
Με τι μοιάζουν αυτές οι εξαρτήσεις;



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
```

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.

Εξωτερικές Παγκόσμιες Μεταβλητές (extern)

Ο προσδιοριστής **extern** μας επιτρέπει να δηλώσουμε και να χρησιμοποιήσουμε μεταβλητές που ορίζονται σε άλλο αρχείο.

err.c

```
int  
error_counter;
```

err.h

```
extern int  
error_counter;
```

init.c

```
#include "err.h"  
...  
error_counter++;
```

Δείκτες σε Συναρτήσεις (Function Pointers)

Όπως έχουμε δείκτες σε δεδομένα, έτσι μπορούμε να έχουμε και δείκτες σε συναρτήσεις. Και ο κώδικας είναι δεδομένα (objdump -d). Γενική μορφή:

τύπος (*όνομα_δείκτη)(λίστα_ορισμάτων) ;

Για παράδειγμα:

```
int (*myprint)(char *message);
```

Διαβάζεται ως: η myprint, είναι ένας δείκτης σε συνάρτηση, όπου αυτή η συνάρτηση έχει τύπο επιστροφής int και ένα όρισμα τύπου char *.

Χρήση Function Pointer

```
#include <stdio.h>

int print_err(char * message) {
    return fprintf(stderr, "%s\n", message);
}

int main() {
    int (*myprint)(char *message);
    myprint = print_err;
    myprint("hello function pointer");
    return 0;
}
```

```
$ gcc -o funcptr funcptr.c
$ ./funcptr
hello function pointer
```

Ισοδύναμα μπορεί να κληθεί και ως
(*myprint)("hello function pointer");

Variadic Συναρτήσεις

Συναρτήσεις που έχουν μεταβαλλόμενο αριθμό ορισμάτων (π.χ., printf) λέγονται **variadic**. Δηλώνουμε τον μεταβλητό αριθμό ορισμάτων χρησιμοποιώντας την έλλειψη (ellipsis): ...

Παραδείγματα:

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

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

Παράδειγμα με Variadic Function

```
#include <stdarg.h>

int sum(int count, ...) {
    int result = 0;
    va_list args;
    va_start(args, count);
    for(int i = 0; i < count; i++)
        result += va_arg(args, int);
    va_end(args);
    return result;
}
int main() {
    return sum(6, 1, 2, 3, 4, 5, 6);
}
```

```
$ ./variadic
$ echo $?
21
```

Χρησιμοποιώντας τις "μαγικές" συναρτήσεις
va_list, va_start, va_arg, va_end μπορούμε να
διατρέξουμε όλα τα ορίσματα (δεν ξέρουμε τι
κάνουν; χρησιμοποιούμε man!)

Binary Formatted Integers (C2X)

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

```
int main() {
```

```
    int i = 0b101010;
```

```
    printf("My lucky number is: %d\n", i);
```

```
    return 0;
```

```
}
```

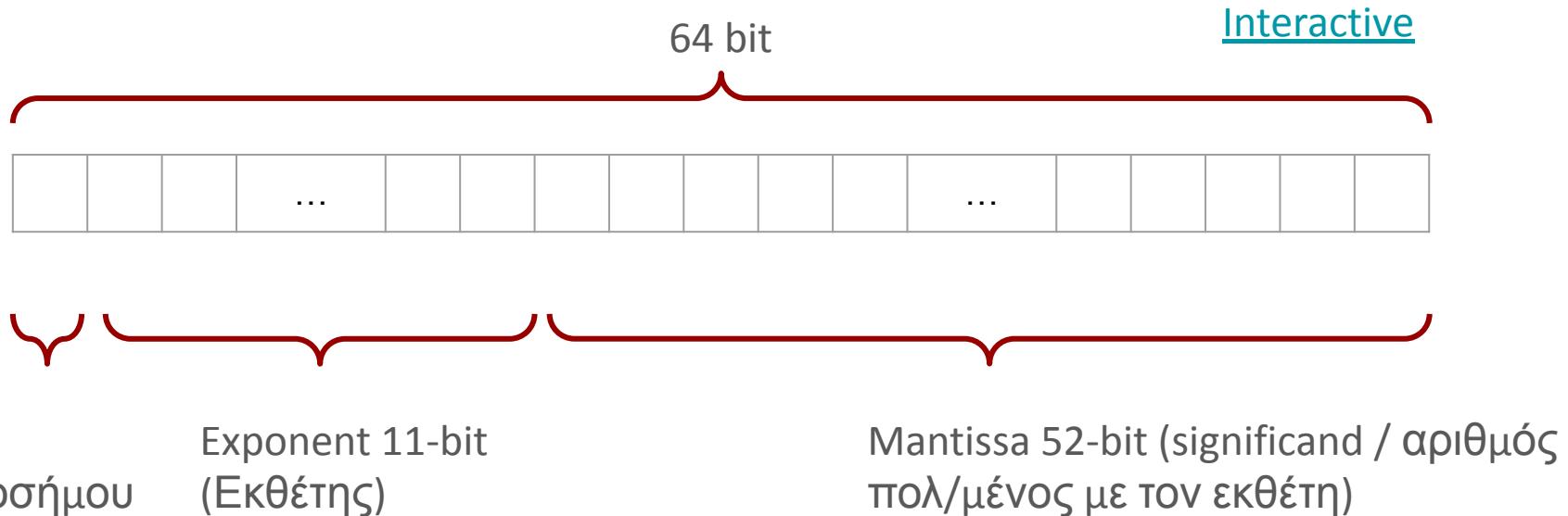
Τι θα τυπώσει αυτό το πρόγραμμα;

```
$ ./bin
My lucky number is: 42
```

Αριθμοί Κινητής Υποδιαστολής

Με 2^n bit μπορούμε και αναπαριστούμε ένα υποσύνολο των πραγματικών αριθμών.

Συνήθως χρησιμοποιούμε το [IEEE754](#) standard (πχ για double 64-bit):



Εκτελώντας άλλα Προγράμματα (exec*, system)

Οποιοδήποτε πρόγραμμα μπορούμε να τρέξουμε από τον φλοιό (shell) του Linux μπορούμε να το τρέξουμε και από ένα πρόγραμμα C. Παραδείγματα:

```
#include <unistd.h>

int main() {
    execl("/bin/ls", "/bin/ls", NULL);
}

#include <stdlib.h>

int main() {
    system("sort -k2 -t, -n");
}
```

Type Qualifier: volatile

Κάθε ανάγνωση/εγγραφή είναι **παρατηρήσιμη**. Ο compiler δεν επιτρέπεται να:

- κρατήσει τιμή σε register
- αφαιρέσει προσπελάσεις
- αναδιατάξει reads/writes

Χρήσιμο όταν η τιμή αλλάζει εκτός ελέγχου
του προγράμματος (memory mapped IO)

```
#define UART_STATUS (*(volatile unsigned int *)0x40001000)

void wait_ready(void) {
    while ((UART_STATUS & 0x1) == 0) {
        /* compiler *must* re-read */
    }
}
```

Type Qualifier: register

Υπόδειξη στον compiler να κρατήσει τη μεταβλητή σε CPU register (καταχωρητή). Απαγορεύει τη λήψη διεύθυνσης (&var) της μεταβλητής.

Σύγχρονοι compilers:

- αγνοούν το register
- better register allocation

Deprecated in C23 :(

```
void sum(const int *a, int n) {  
    register int i;  
    int s = 0;  
    for (i = 0; i < n; i++)  
        s += a[i];  
}
```

Type Qualifier: restrict (since C99)

Μια υπόδειξη προς τον compiler πως ένας δείκτης είναι ο μοναδικός τρόπος πρόσβασης στο αντικείμενο και καμία άλλη αναφορά δεν δείχνει στην ίδια μνήμη.

```
char * strcpy(char * restrict s1, const char * restrict s2);
```

```
int main() {
    char s[50] = "hello world";
    strcpy(s, s);
    return 0;
}
```

```
$ gcc -Wrestrict -o main main.c
main.c: In function 'main':
main.c:4:12: warning: passing argument 1 to
'restrict'-qualified parameter aliases with
argument 2 [-Wrestrict]
        4 |     strcpy(s, s);
          |           ^ ~
```

Type Qualifier: auto

Δηλώνει automatic storage duration και ισχύει για μεταβλητές μέσα σε blocks.

Είναι ήδη το default!

```
void foo() {  
    auto int x = 42; /* equivalent to: int x = 42; */  
    printf("%d\n", x);  
}
```

| | | | | |
|---------------------|-------------------|---------------------|--------------------|---------------------|
| 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 | |

Debugging με GDB

1. gcc -g -ggdb -o prog prog.c
2. gdb --args ./program arg1 arg2
3. run, break, step, continue, finish
4. backtrace
5. print
6. [Cheat Sheet](#)

Αναφορές

Από τις διαφάνειες του κ. Σταματόπουλου προτείνω να διαβάσετε τις σελίδες 64-68, 104, 167-168

- [Lines of Code Written](#)
- [Variadic Function, simple_printf, va_start](#)
- [const και extern](#)
- [Function Pointer](#)
- [exec](#)
- [IEEE754](#)
- [Type Qualifiers](#)

Ευχαριστώ και Καλά Χριστούγεννα εύχομαι!
Keep Coding ;)