

Διάλεξη 23 - Προχωρημένα Θέματα

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

Θανάσης Αυγερινός

Ανακοινώσεις / Διευκρινίσεις

- Η Εργασία #2 πρέπει να υποβληθεί μέχρι την Τρίτη 14/1/2025, 23:59 - μην το ξεχάσουμε!
- Βγήκε η Εργασία #3 - Προθεσμία: 26 Φεβρουαρίου 2025, 23:59
- Ημερομηνία εξέτασης μαθήματος: 20 Ιανουαρίου 2025
- Ημερομηνία εξέτασης εργαστηρίων: σε συνεννόηση με τους υπευθύνους εργαστηρίων
- Η προφορική εξέταση των εργασιών θα γίνει με συνεννόηση με τα άτομα που θα σας εξετάσουν, περιμένετε email μέσα στις επόμενες δύο εβδομάδες
- Την επόμενη Παρασκευή (17/1), φροντιστηριακό μάθημα για τις εξετάσεις

Πέρσι

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



Φέτος

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

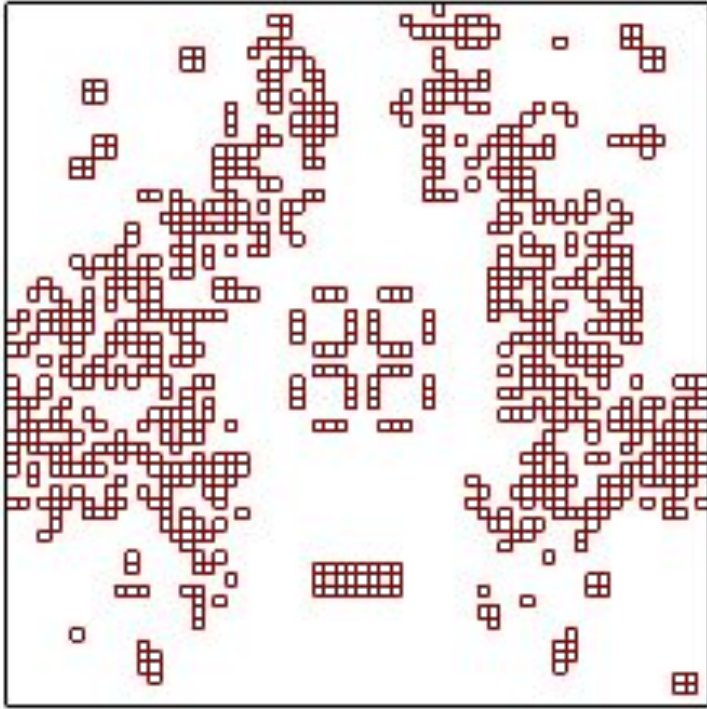


Γραμμές Κώδικα (Lines of Code - LOC)

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



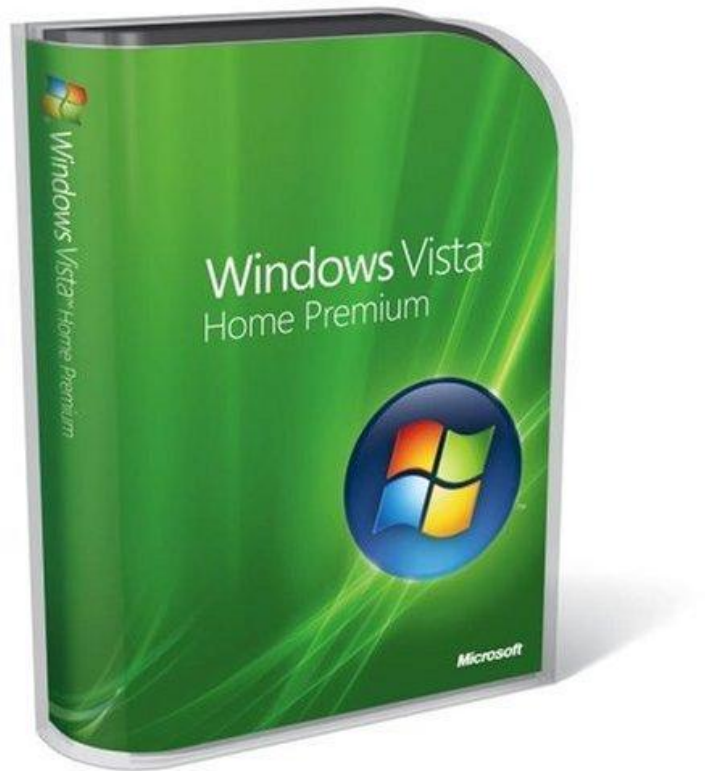
Conway's Game of Life (1970)

~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

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
```

```
gcc -c -o init.o init.c
```

```
gcc -o out init.o err.o
```

err.o

out

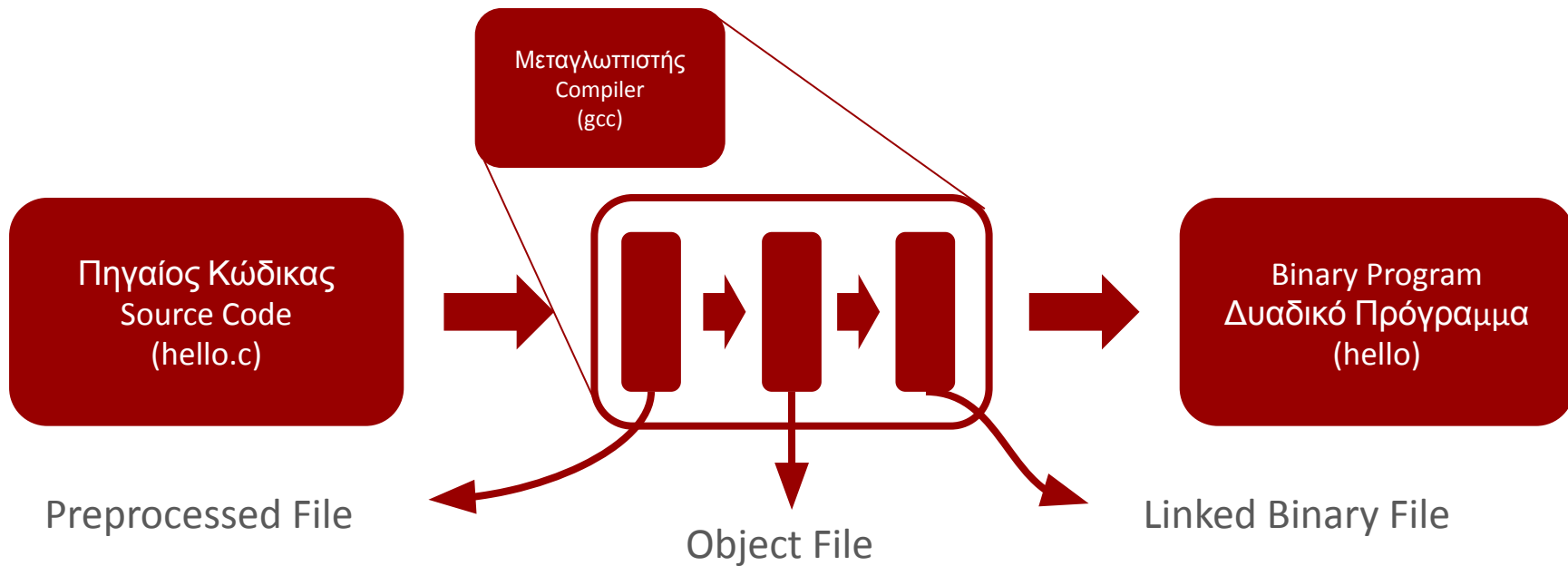
init.o

T print_err

T print_err

U print_err

Μεταγλώττιση

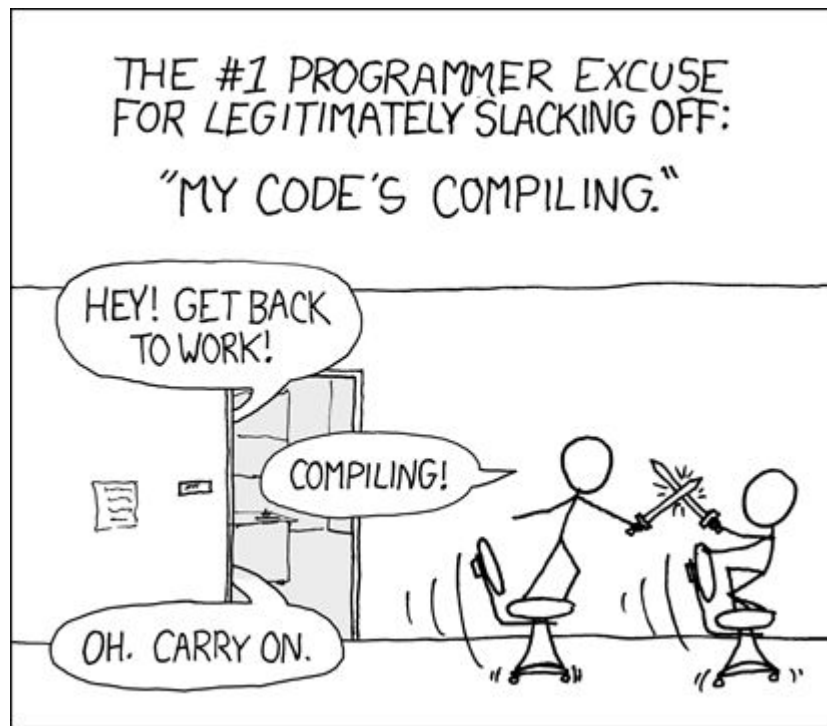


Η Μεταγλώττιση είναι Χρονοβόρα Διαδικασία

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 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 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.

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++;
```

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;

}
```

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.bin", "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.bin", "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

6c6c6568

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;
}
```

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.bin", "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;

}
```

Παράδειγμα: Γράψιμο Αρχείων

```
#include <stdio.h>

int main() {

    FILE *fileToWrite;

    fileToWrite = fopen("output.bin", "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...|
```

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;
```

```
}
```

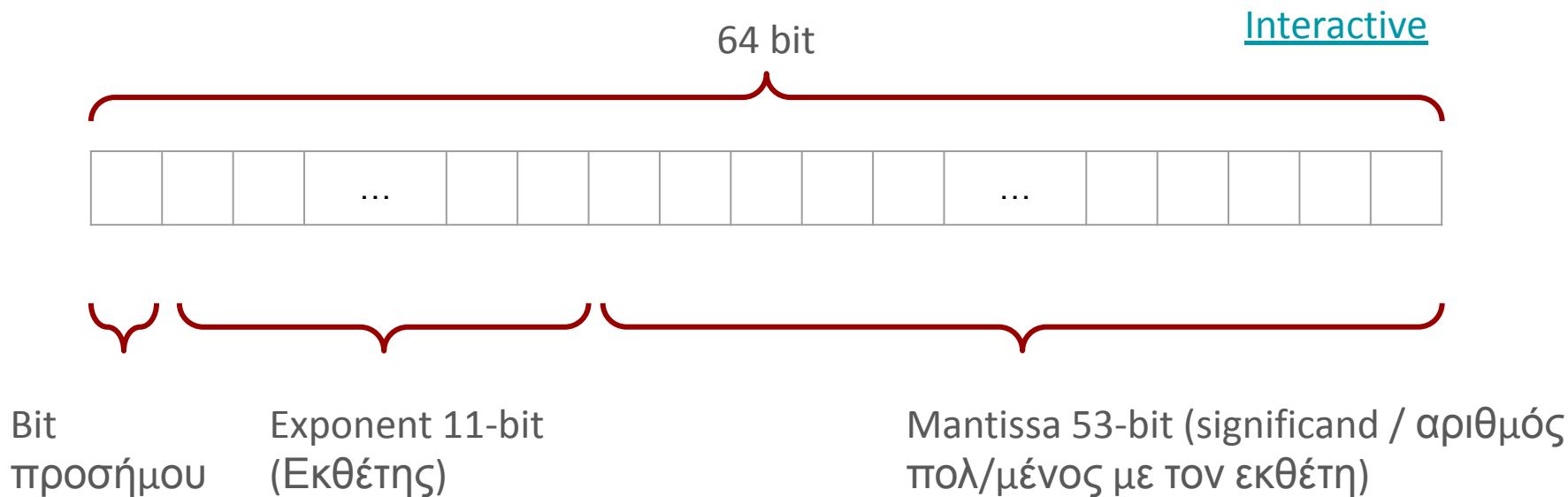
Τι θα τυπώσει αυτό το πρόγραμμα;

```
$ ./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");
```

```
}
```

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](#)

Ευχαριστώ και καλό σαβ/κο εύχομαι!
Keep Coding ;)