

Διάλεξη 11 - Δείκτες και Αναδρομή

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

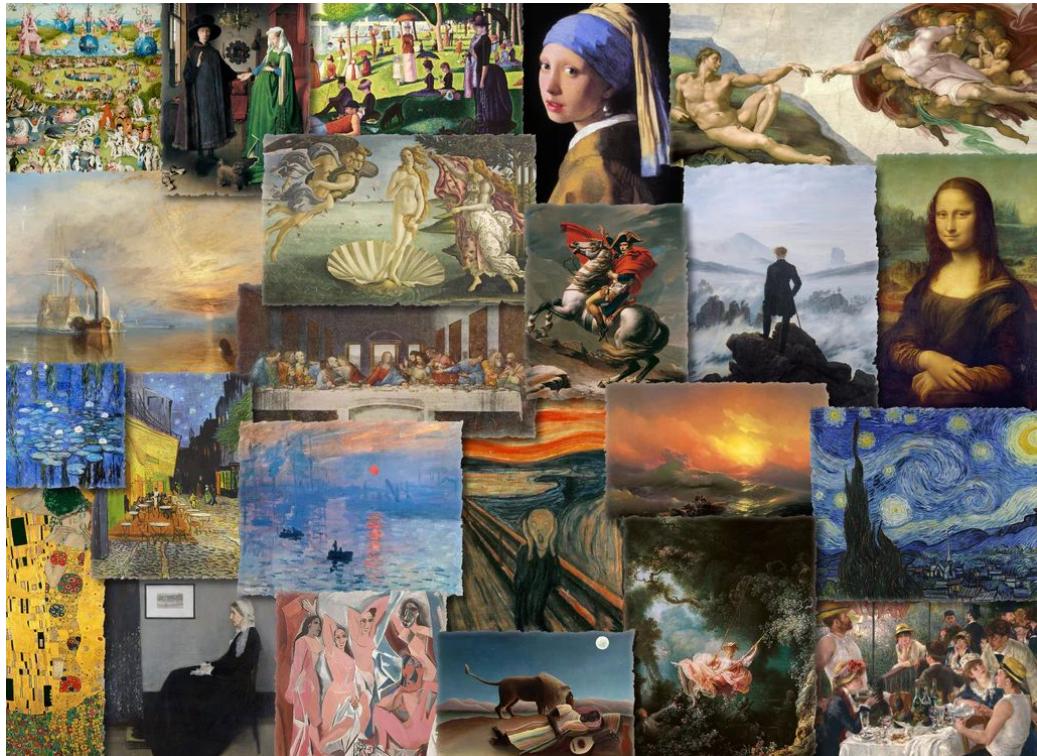
Θανάσης Αυγερινός / Τάκης Σταματόπουλος

Ανακοινώσεις / Διευκρινίσεις

- Η Εργασία #1 έρχεται την επόμενη εβδομάδα!

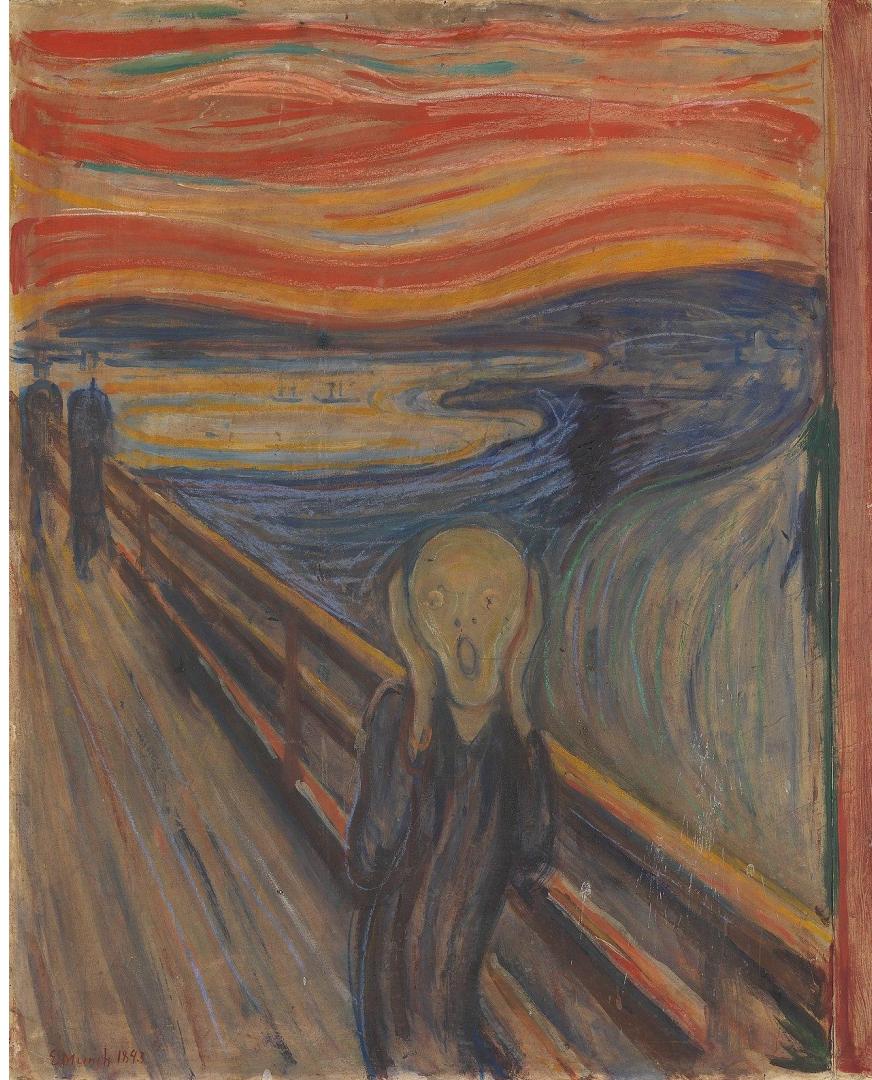
Την προηγούμενη φορά

- Λίγο για scanf
- Πίνακες (Arrays)



Σήμερα

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



Η Μνήμη Οργανώνεται σε Bytes (Υπενθύμιση)

Το μέγεθος της μνήμης μετράται σε Bytes:

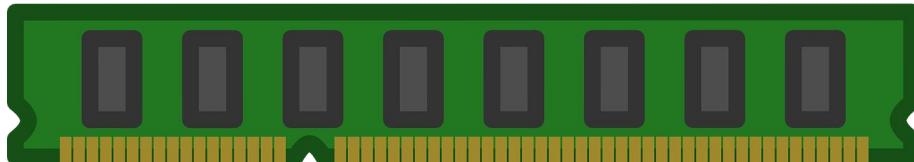
- **1 KB (KiloByte) = 1.000 Bytes**
- **1 MB (MegaByte) = 1.000.000 Bytes**
- **1 GB (GigaByte) = 1.000.000.000 Bytes**

Μνήμη με
N Bytes

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

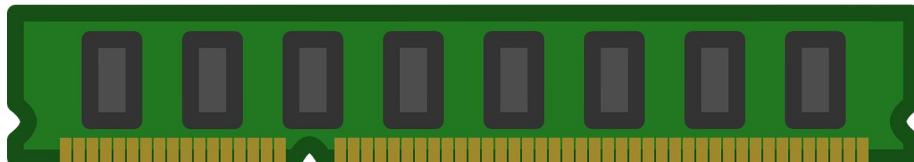
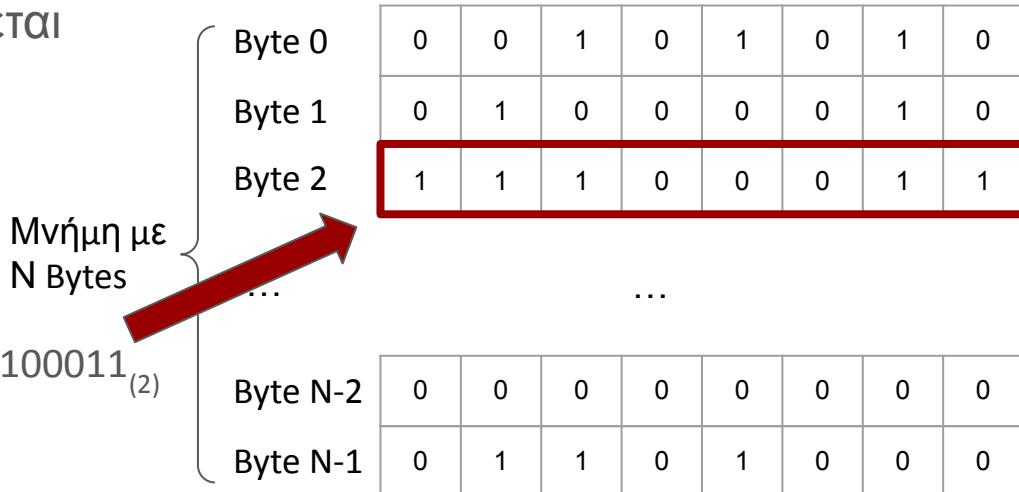
0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
1	1	1	0	0	0	1	1
...							

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).



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 μεταγλωπιστή πόση μνήμη να δεσμεύσει για τα περιεχόμενα

Το όνομα (name) της μεταβλητής κάνει τον μεταγλωπιστή 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-2	0	0	0	0	0	0	0	0
Byte N-1	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-2	0	0	0	0	0	0	0	0
Byte N-1	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-2	0	0	0	0	0	0	0	0
Byte N-1	0	1	1	0	1	0	0	0

Διεύθυνση (Address) Μιας Μεταβλητής

Μπορούμε να βρούμε την **διεύθυνση** μιας μεταβλητής χρησιμοποιώντας 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 χρησιμοποιώντας 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

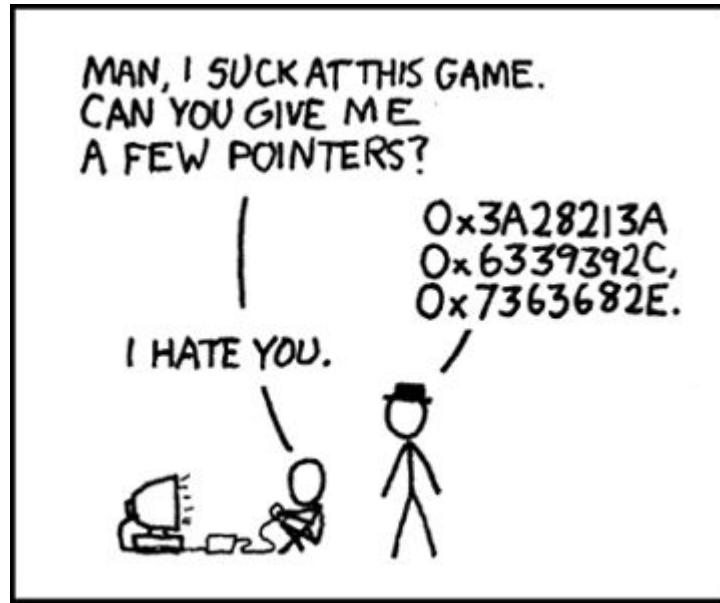
...

...

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

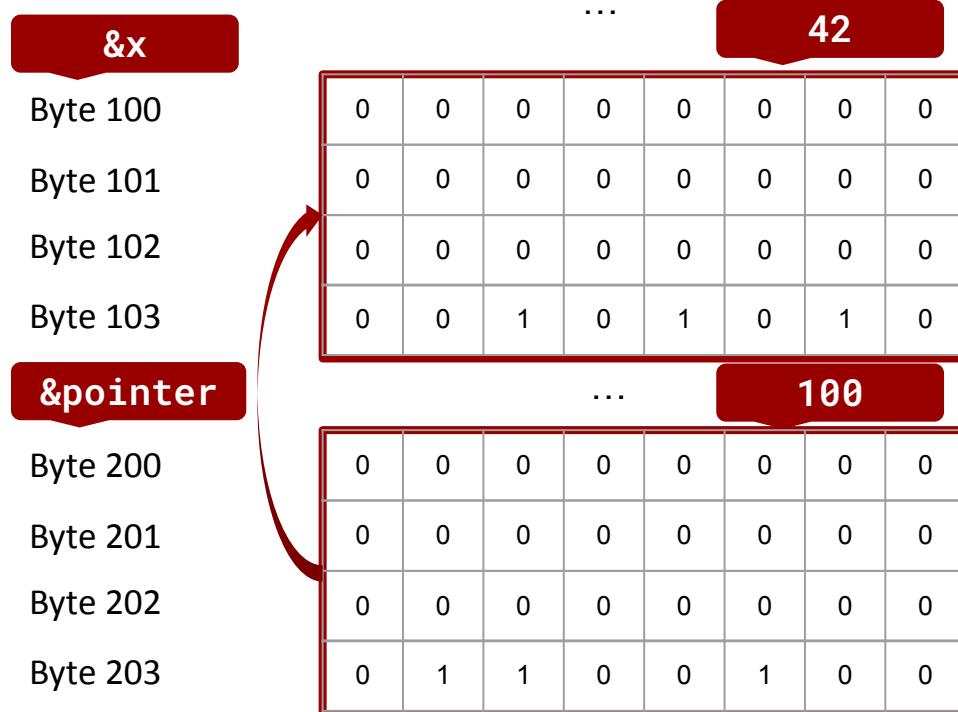
Αρχικοποιούμε έναν δείκτη με την διεύθυνση μιας μεταβλητής ως εξής:

```
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

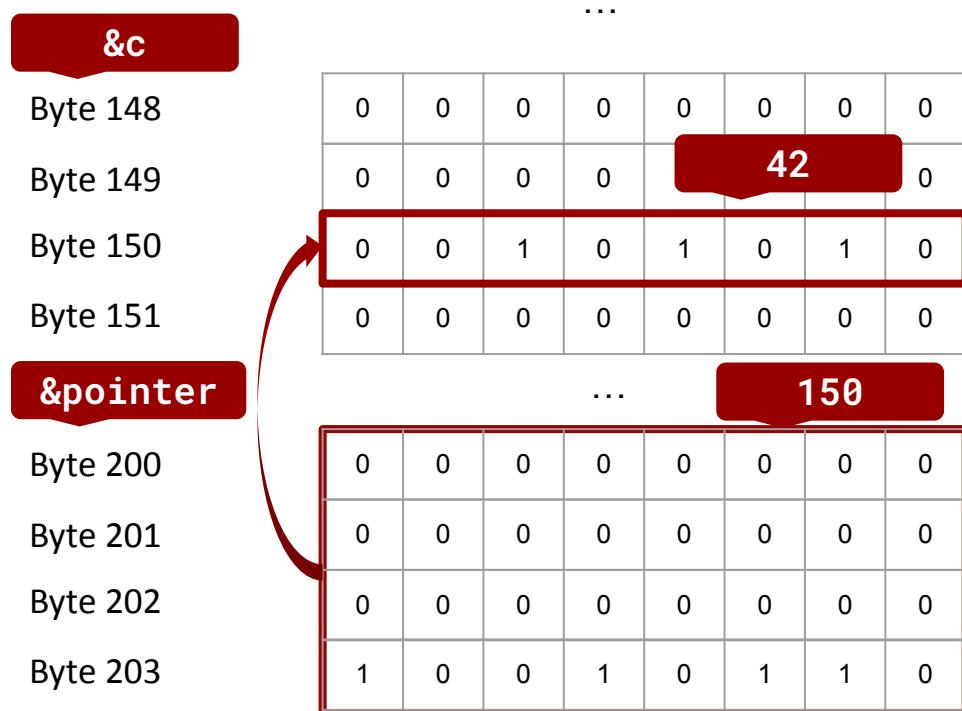
Αρχικοποιούμε έναν δείκτη με την διεύθυνση ενός χαρακτήρα ως εξής:

```
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 * :

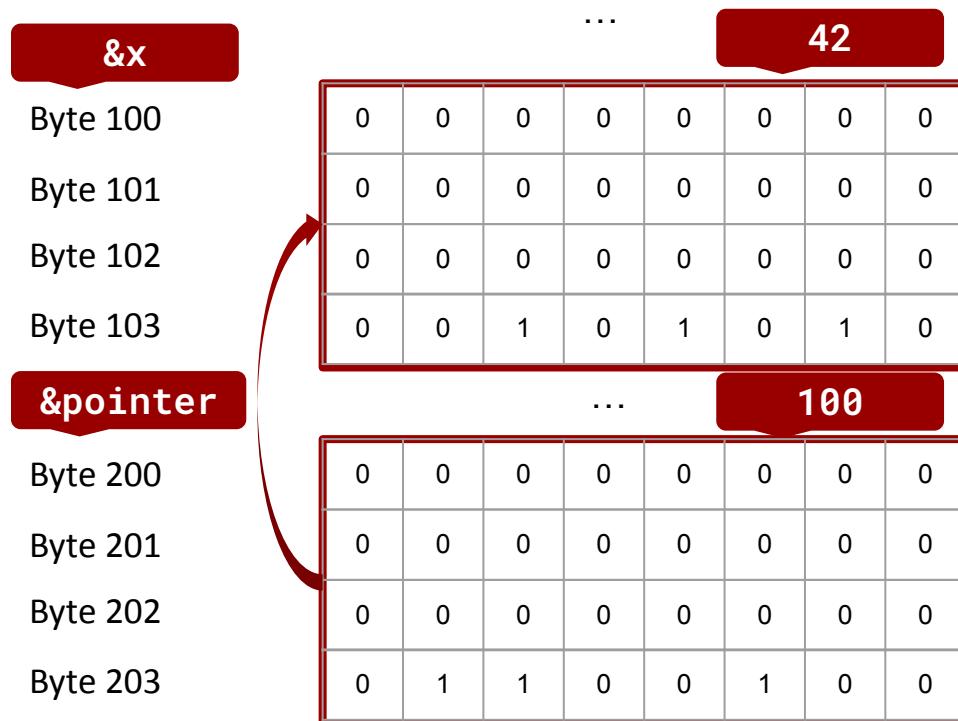
```
int x = 42;  
int *pointer = &x;  
printf("%d\n", *pointer);
```

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 αλληλοαναιρούνται 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

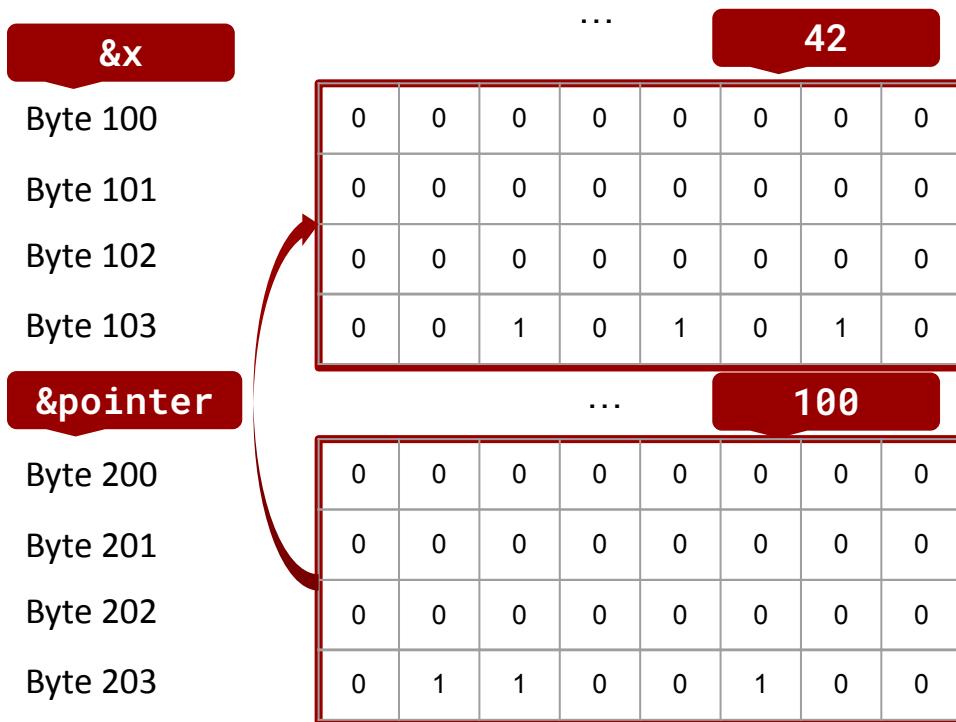
Για να χρησιμοποιήσουμε το περιεχόμενο της μεταβλητής στην οποία δείχνει ένας δείκτης χρησιμοποιούμε τον μοναδιαίο τελεστή * :

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

Όταν το τρέξουμε:

```
$ ./test
```

```
104
```



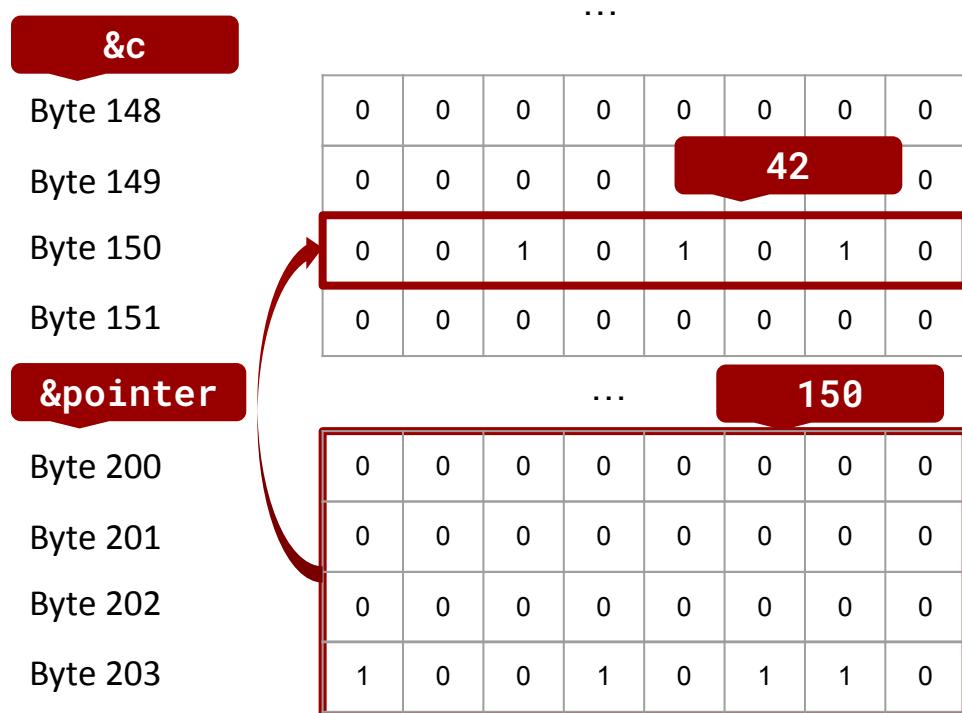
Μείωση δείκτη σε char

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 $*(\text{ptr} + n)$ - υπολόγισε την διεύθυνση του δείκτη μετά από n στοιχεία και επίστρεψε την μεταβλητή 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 $*(\text{ptr} + n)$ - υπολόγισε την διεύθυνση του δείκτη μετά από n στοιχεία και επίστρεψε την μεταβλητή 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;
```

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

Από την προηγούμενη φορά, πως μπορώ να τυπώσω "World\n";

```
char hello[] = "Hello World\n";
```

```
char *world = ...;
```

```
printf("%s", world);
```

Από την προηγούμενη φορά, πως μπορώ να τυπώσω "World\n";

```
char hello[] = "Hello World\n";
char *world = &hello[6];
printf("%s", world);
```

Παραδείγματα με Πίνακες

Θέλω μια συνάρτηση που να δέχεται έναν πίνακα 100 ακεραίων και να επιστρέφει τον μέσο όρο. Πως;

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;

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 atoι που να παίρνει ένα πίνακα χαρακτήρων (μόνο ψηφία) και να επιστρέψει έναν ακέραιο. Πως;

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;

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)**:

$$n! = \begin{cases} 1, & \text{if } n = 0, \\ n \times (n - 1)!, & \text{if } n > 0. \end{cases}$$

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 == 0) 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;
}

```

$$n! = \begin{cases} 1, & \text{if } n = 0, \\ n \times (n - 1)!, & \text{if } n > 0. \end{cases}$$



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

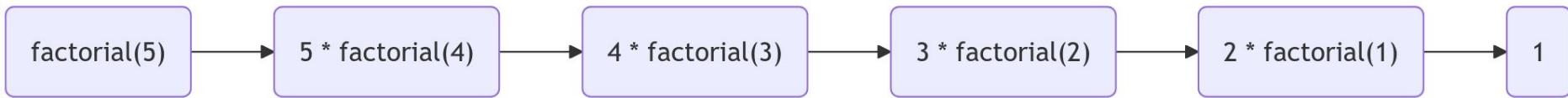
Παρατηρήσεις

- 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)

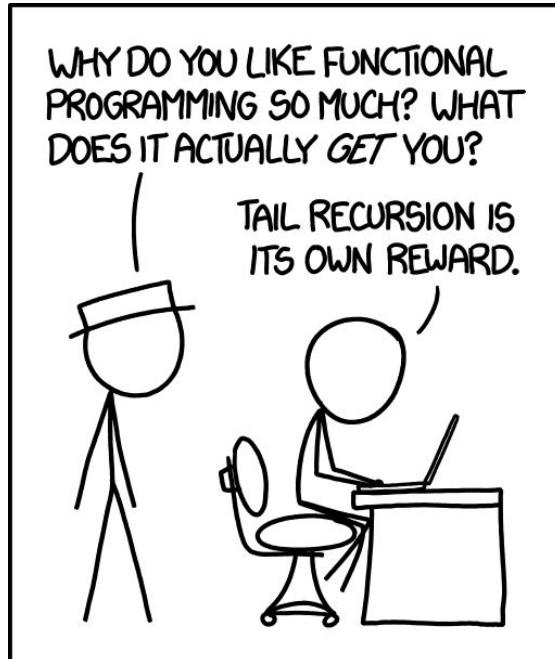
Αναδρομή είναι η μέθοδος κατά την οποία μια συνάρτηση **καλεί τον εαυτό της**, με στόχο να επιλύσει ένα υποπρόβλημα του αρχικού προβλήματος, έως ότου φτάσει σε μια βάση τερματισμού (base case) όπου η αναδρομή σταματά.

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

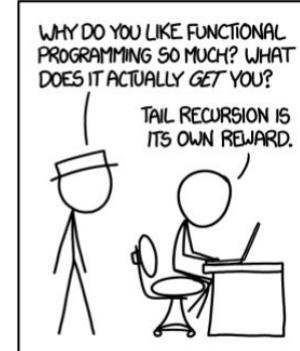
Η Αναδρομή Πρέπει να Τερματίζει

Δεν ξεχνάμε να γράψουμε σωστά Base Cases



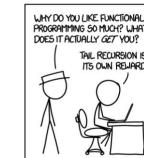
Η Αναδρομή Πρέπει να Τερματίζει

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 ;)