

# Διάλεξη 13 - Μνήμη

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στον Προγραμματισμό

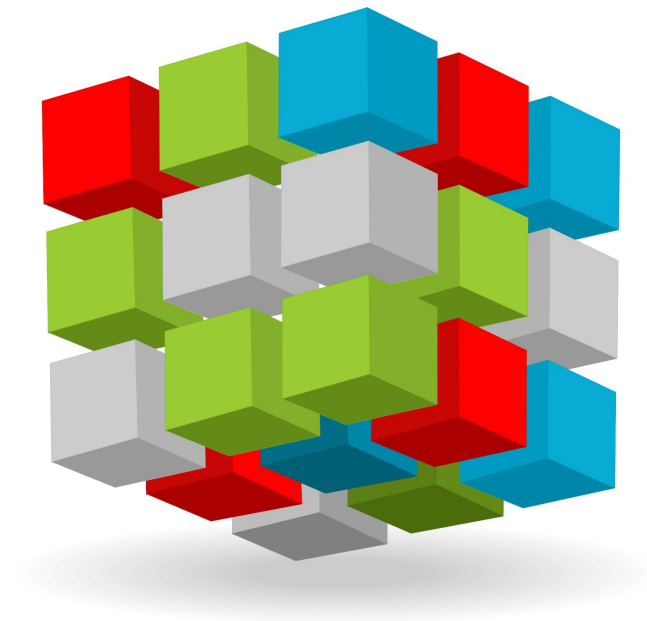
Θανάσης Αυγερινός / Τάκης Σταματόπουλος

## Ανακοινώσεις / Διευκρινίσεις

- Η Εργασία #0 τελείωσε!
- Βγήκε η Εργασία #1. Προθεσμία: *Wednesday, 10th of December 23:59*
  - Συζήτηση
- Γρίφοι & bonus: μια ευκαιρία για εξάσκηση

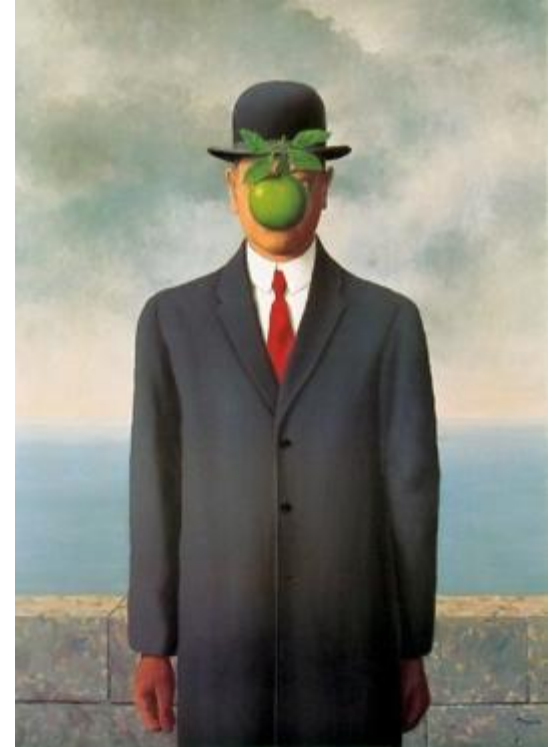
# Την Προηγούμενη Φορά

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



# Σήμερα

- Endianness
- Διαχείριση Μνήμης
- Κατηγορίες Μνήμης
- Συναρτήσεις, Μεταβλητές,  
Δείκτες, Πίνακες
- Παραδείγματα



# 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).

|          |          |          |          |
|----------|----------|----------|----------|
| 68 ('h') | 65 ('e') | 6c ('l') | 6c ('l') |
|----------|----------|----------|----------|

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 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

...

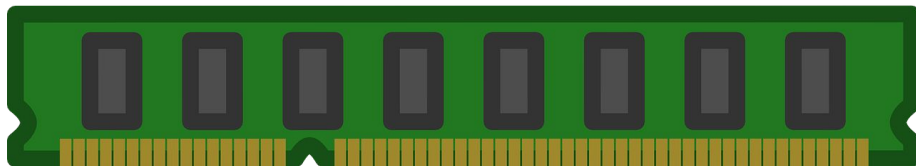
Byte N-2

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 |





# 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

Byte 31999

Byte 32000

63

62

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

Byte 31999

Byte 32000

64

63

62

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;
```

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

Byte 32000

62

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

# 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

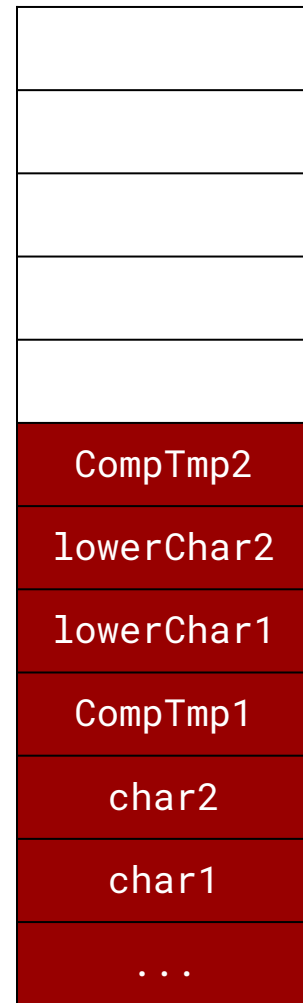
3: Προσωρινά δεδομένα (συνήθως  
μερικά bytes) που αποθηκεύει ο  
μεταγλωττιστής

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



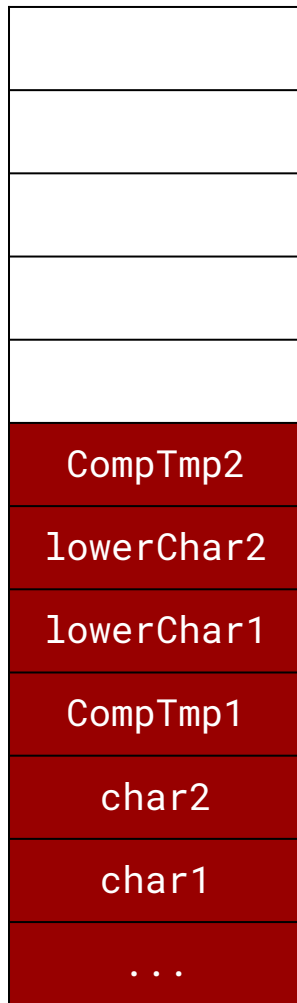
## Τι αποθηκεύεται συνήθως στην στοίβα;

```
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



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;  
}
```

tolower

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

equalIgnoreCase

CompTmp5

CompTmp4

CompTmp3

c

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

## 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;
```

```
}
```

tolower

equalIgnoreCase

CompTmp5

CompTmp4

CompTmp3

c

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

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;
```

```
}
```

tolower

equalIgnoreCase

CompTmp5

CompTmp4

CompTmp3

c

CompTmp2

lowerChar2

lowerChar1

CompTmp1

char2

char1

...

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();  
}
```

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

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

## 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

Τι θα κάνει το ακόλουθο πρόγραμμα;

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

```
int main() {
```

```
    char bomb[9000000];
```

```
    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)

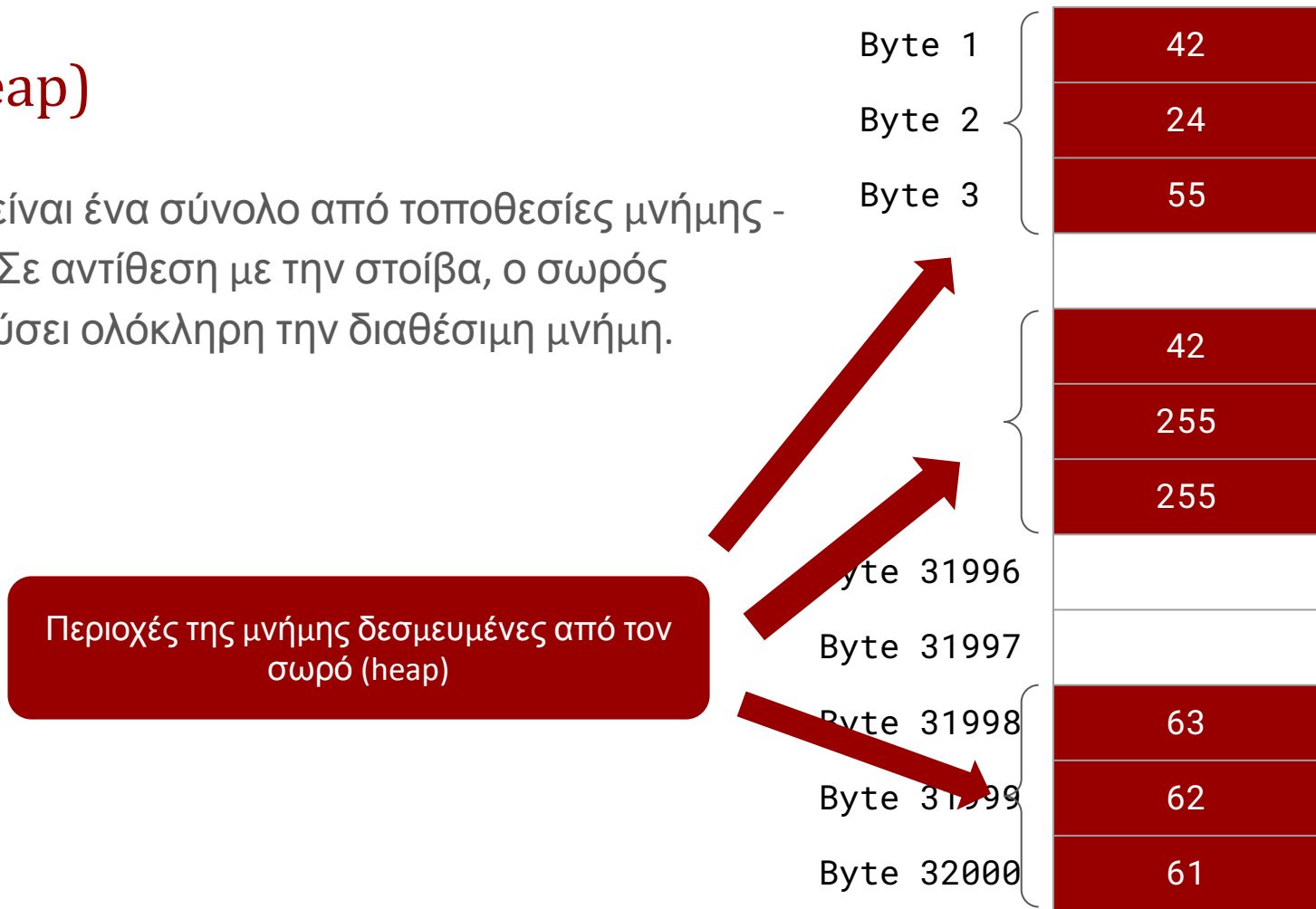
Ο **σωρός (heap)** είναι ένα σύνολο από τοποθεσίες μνήμης - σε τυχαία σειρά. Σε αντίθεση με την στοίβα, ο σωρός μπορεί να δεσμεύσει ολόκληρη την διαθέσιμη μνήμη.





# 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.



# 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(τυπος));**

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:

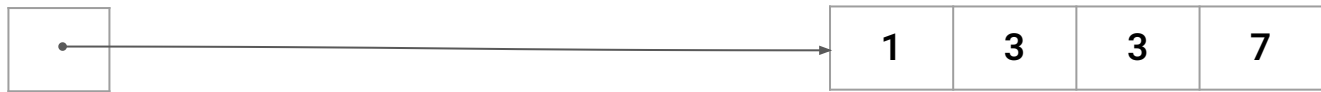
```
int * array = malloc(4 * sizeof(int));
```

stdlib.h

Δημιουργία  
πίνακα 4 ακεραίων

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

```
int * array
```



array[0]

array[3]

# 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 στον **σωρό (heap)**

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));
```

## 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
```

# 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]`

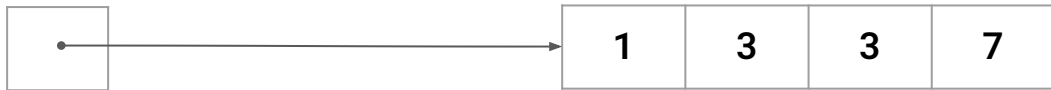
`array[3]`

## Δέσμευση Μνήμης σωρού με malloc

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

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

```
int * array
```

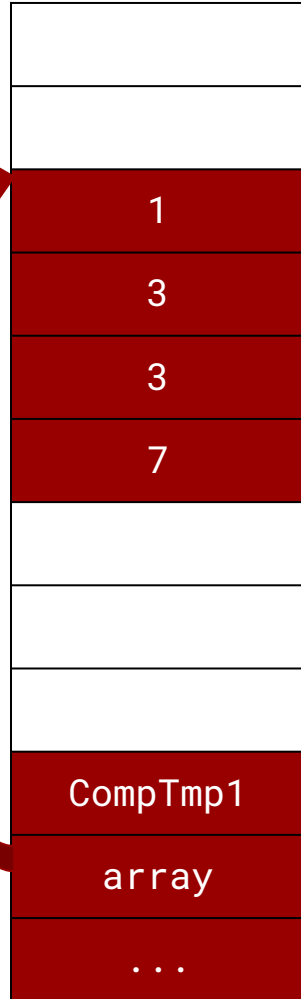


array[0]

array[3]

heap

stack





## 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 αποτέλεσμα τους μπορεί να χρησιμοποιηθεί ως δείκτης σε `int`, `double`, `char` - ανάλογα με την χρήση. Όλα είναι δείκτες ίδιου μεγέθους.

Τι 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
```

Τι θα κάνει το ακόλουθο πρόγραμμα;

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

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

```
int main() {
```

```
    char *bomb = malloc(sizeof(char) * 9000000000000L);
```

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

```
    return 0;
```

```
}
```

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

Τι θα κάνει το ακόλουθο πρόγραμμα;

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

int main() {
    char *bomb = malloc(sizeof(char) * 9000000000000L);
    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) * 900000000000L);
    if (!bomb) {
        fprintf(stderr, "array allocation failed\n");
        exit(1);
    }

    bomb[0] = 'a';

    printf("Hello World\n");

    return 0;
}
```



Αν δεν υπάρχει αρκετή μνήμη, η malloc μας επιστρέφει 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);
```

Ή ισοδύναμα

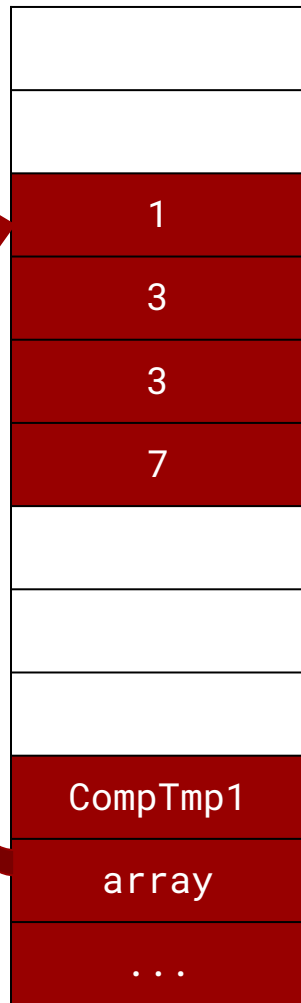
array == NULL

Φροντίζουμε κάθε  
κλήση malloc να  
συνοδεύεται από  
μια free

Τι θα συμβεί στην μνήμη μετά την free;

heap

stack

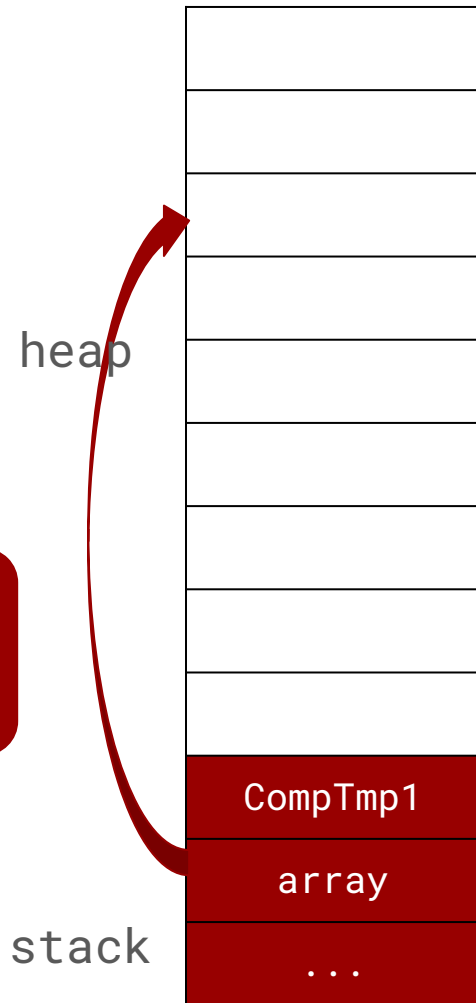


## Αποδέσμευση Μνήμης με free

```
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;

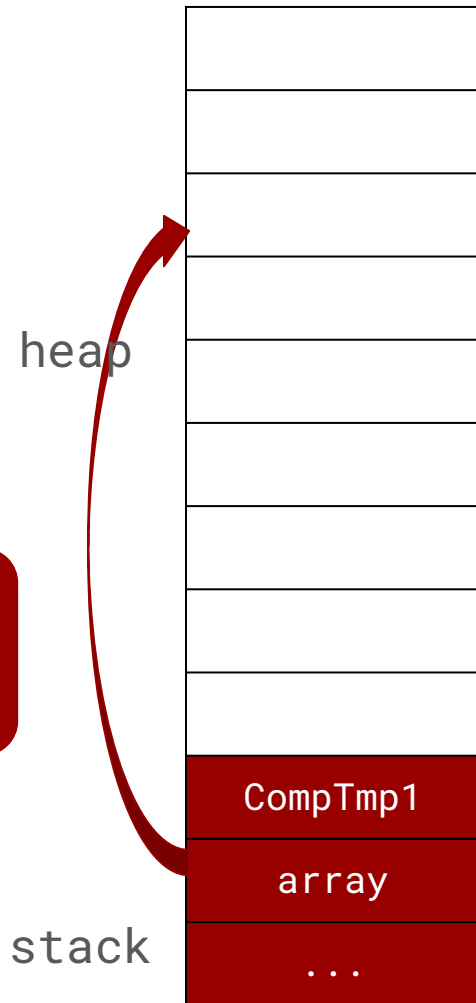


## Αποδέσμευση Μνήμης με free

```
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 αποδεσμεύσω μνήμη που έχω αποδεσμεύσει;



# 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:

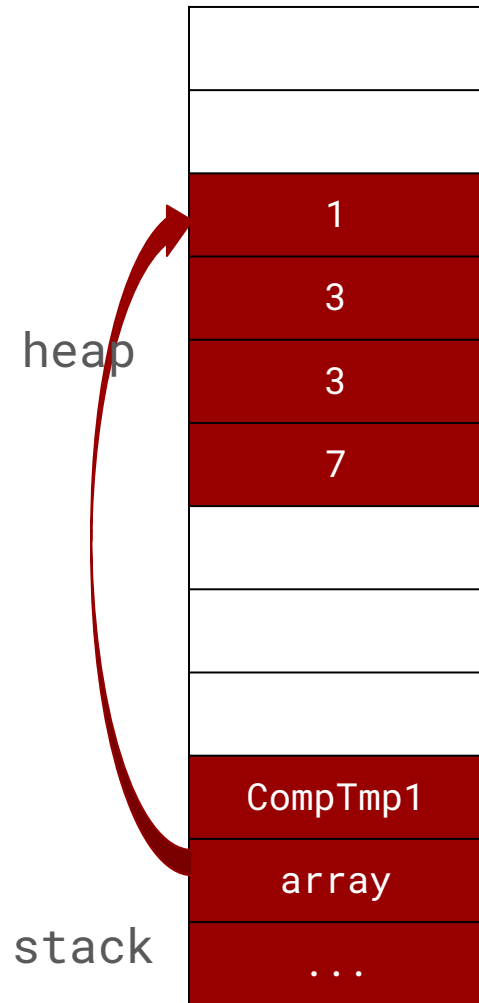
```
int * array = malloc(4 * sizeof(int));  
if (!array) {...}  
array = realloc(array, 8192 * sizeof(int));  
if (!array) {...}  
array[8000] = 42;  
free(array);
```

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);
```



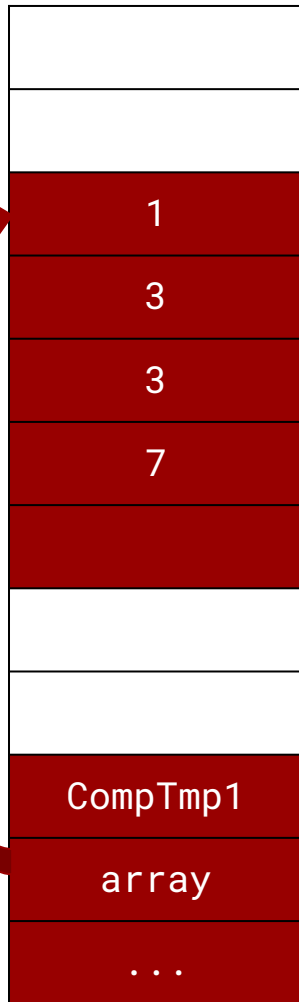
## Αλλαγή μεγέθους με την `realloc`

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

Τα περιεχόμενα της  
μνήμης  
διατηρούνται

heap

stack



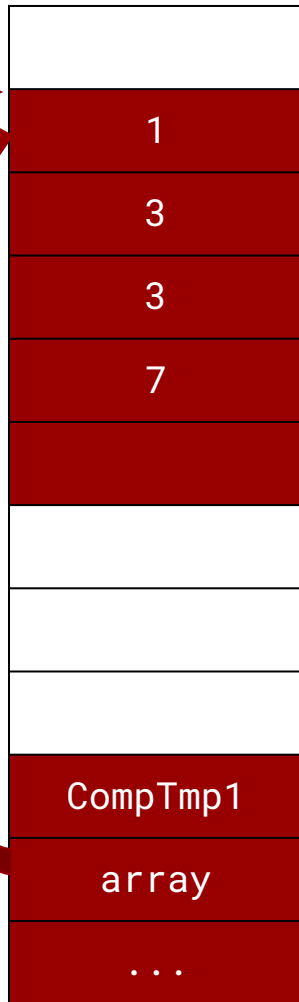
# Αλλαγή μεγέθους με την `realloc`

Η `realloc` μπορεί να αλλάξει την αρχική διεύθυνση μέσα στον σωρό

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

heap

stack



Θέλω να δημιουργήσω έναν δισδιάστατο πίνακα ακεραίων  $M \times N$  δυναμικά. Πως;

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

if (!array) {
    fprintf(stderr, "array allocation failed\n");
    exit(1);
}

for(int i = 0 ; i < M ; i++) {
    array[i] = malloc(N * sizeof(int));

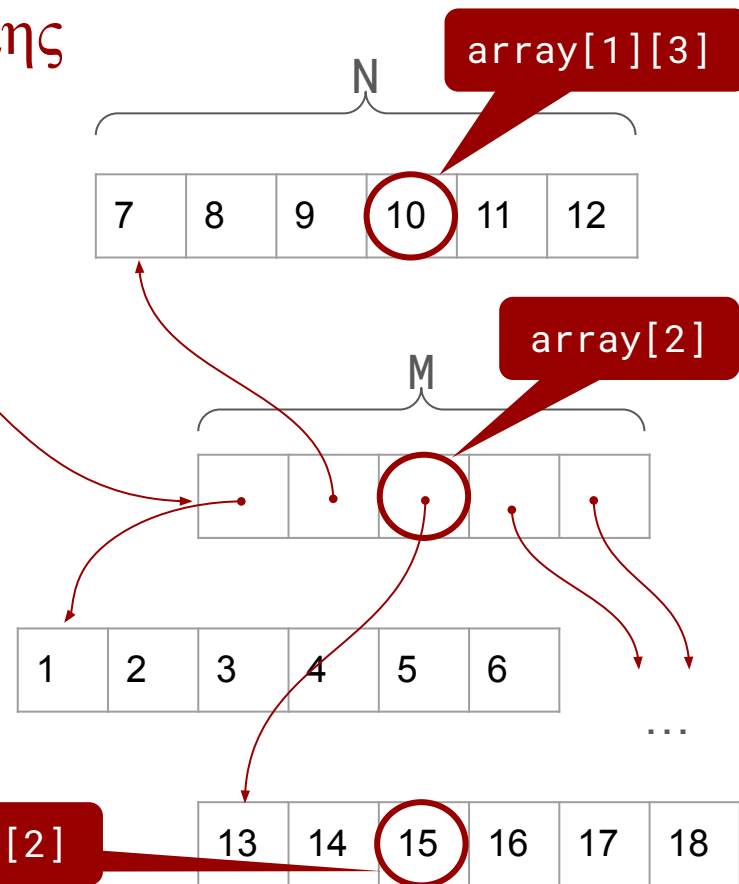
    if (!array[i]) {
        fprintf(stderr, "array[%d] alloc failed\n", i);
        exit(1);
    }
}
```



## Έστω $M = 5$ , $N = 6$ - Περιεχόμενα Μνήμης

```
int ** array = malloc(M * sizeof(int*));  
if (!array) {  
    fprintf(stderr, "array allocation failed\n");  
    exit(1);  
}  
for(int i = 0 ; i < M ; i++) {  
    array[i] = malloc(N * sizeof(int));  
    if (!array[i]) {  
        fprintf(stderr, "array[%d] alloc failed\n", i);  
        exit(1);  
    }  
}
```

array



Θέλω να αποδεσμεύσω την μνήμη από έναν δισδιάστατο πίνακα  
ακεραίων  $M \times N$  δυναμικά. Πως;

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

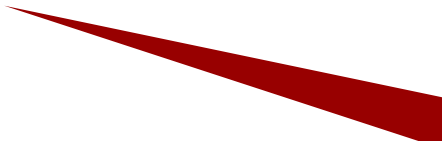
```
...
```

```
for(int i = 0 ; i < M ; i++) {
```

```
    free(array[i]);
```

```
}
```

```
free(array);
```



Αποδεσμεύουμε **πρώτα**  
τους υποπίνακες και μετά  
τον πίνακα δεικτών

# Για την επόμενη φορά

Αν θέλετε να μάθετε περισσότερα για διαχείριση μνήμης:

- [Memory management and heap](#)
- [Stack memory](#)
- [Activation records](#)
- [Dynamic memory allocation in C](#)
- [Memory leaks](#)
- [Data segment](#) and [bss section](#) (another time)
- [Endianness](#)

Ευχαριστώ και καλή μέρα εύχομαι!  
Keep Coding ;)

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);  
}
```

Θέλω να δημιουργήσω έναν δισδιάστατο πίνακα ακεραίων  $M \times N$  δυναμικά. Πως;

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

if (!array) {
    perror("array allocation");
    exit(1);
}

for(int i = 0 ; i < M ; i++) {
    array[i] = malloc(N * sizeof(int));

    if (!array[i]) {
        perror("array[i] failed");
        exit(1);
    }
}
```