



**Τμήμα Μηχανικών Η/Υ & Πληροφορικής  
Πανεπιστήμιο Πατρών**

# Εισαγωγή στον Προγραμματισμό

---

Η γλώσσα προγραμματισμού C

# Δείκτες

---

- Η C δίνει τη δυνατότητα να αποθηκεύουμε **διευθύνσεις μνήμης** σε ειδικές μεταβλητές που λέγονται **pointers**

```
int *i;      // pointer to int
char *c;     // pointer to char
double *d;   // pointer to double
```

- Ο κάθε pointer είναι associated με κάποιον τύπο δεδομένων.
  - Π.χ., το `i` στο παραπάνω παράδειγμα δείχνει σε `int`
- Ορίζονται με ένα αστεράκι πριν το όνομα της μεταβλητής
  - Π.χ., στο παρακάτω, μόνο το `p` είναι pointer

```
int i, a[100], *p;
```

# Τελεστές Δεικτών

---

## □ Τελεστής διεύθυνσης (address operator): **&**

- Επιστρέφει τη διεύθυνση μιας μεταβλητής

```
int i, *p;  
p = &i;    // p = address of i
```

## □ Τελεστής αποαναφοράς (dereference operator): **\***

- Μας επιτρέπει να προσπελάσουμε το περιεχόμενο της μνήμης που δείχνει ένας pointer

```
// συνέχεια από το προηγούμενο  
*p = 42;  
printf("%d\n", i);    // 42  
i /= 2;  
printf("%d\n", *p);    // 21
```

# Τελεστές Δεικτών

---

## □ Τελεστής ανάθεσης: =

```
int i=5, *p, *q;  
p = &i;    // p = address of i  
q = p;  
printf("%d\n", *q);    // 5
```

## □ Προσοχή! Άλλο η ανάθεση δείκτη, κι άλλο η ανάθεση της τιμής που δείχνει ο δείκτης

```
q = p;    // Ανάθεση δείκτη  
*q = *p;  // Ανάθεση τιμής
```

# Παράδειγμα

---

```
int x=1;
int y=2;
int *p;
p = &x; /* η p δείχνει τη διεύθυνση της x */
y = *p; /* η y γίνεται 1, δηλαδή η τιμή της x */
*p = 0; /* η x γίνεται 0 */
*p = *p + 10; /* x=x+10 */

*p += 1; /* x=x+1 */
++*p;    /* x=x+1 */
(*p)++;  /* x=x+1 */
```

# Παράδειγμα

---

```
#include <stdio.h>

main()
{
    int x=5; int y=10; int *ptr; int **ptr2;
    ptr=&x;
    ptr2=&ptr;
    *ptr2=&y;
    printf("%d", *ptr);
}
```

# Τελεστές Δεικτών

- Οι τελεστές **&** και **\*** είναι «αντίστροφοι», συνεπώς εν γένει αλληλοαναιρούμενοι

```
int i, j, *p=NULL, *q=&i;  
i = *&j;    // i=j  
i = *&j;    // compile error! (j not ptr)  
p = *&q;    // p=q
```

- Γενικά είναι καλό να αρχικοποιείτε πάντα τους pointers (π.χ., με NULL), αλλιώς μπορεί να «δείχνουν» σε αυθαίρετη διεύθυνση με ολέθρια αποτελέσματα!
  - Τι είναι το NULL???
  - Είναι ειδική τιμή (συγκεκριμένα η τιμή 0), που σημαίνει τον **κενό δείκτη**, δηλαδή έναν pointer που δεν δείχνει πουθενά
  - Ο NULL pointer δεν αποδεικτοδοτείται. Π.χ., το `p=NULL`; `*p = 5`; θα δημιουργήσει λάθος, δεν θα βάλει το 5 στη διεύθυνση 0.

# NULL

---

## □ Τι είναι το **NULL**???

- Είναι ειδική τιμή (συγκεκριμένα η τιμή 0), που σημαίνει τον **κενό δείκτη**, δηλαδή έναν pointer που δεν δείχνει πουθενά
- Ο NULL pointer δεν αποδεικτοδοτείται
- Π.χ., ο παρακάτω κώδικας θα δημιουργήσει λάθος, δεν θα βάλει το 5 στη διεύθυνση 0

```
int *p = NULL;  
*p = 5;
```

## □ Και τι είναι το **void \***???

- Είναι ο τύπος δεδομένων γενικού δείκτη (δείκτη σε κενό)
- Δηλαδή δείκτης που δεν είναι associated με συγκεκριμένο τύπο (int, char, κ.τ.λ.)
- Κανονικός δείκτης, απλά δεν μπορεί να κάνει **pointer arithmetic**
- Μα, τι είναι pointer arithmetic??? 😊



# Pointer Arithmetic (Αριθμητική Δεικτών)

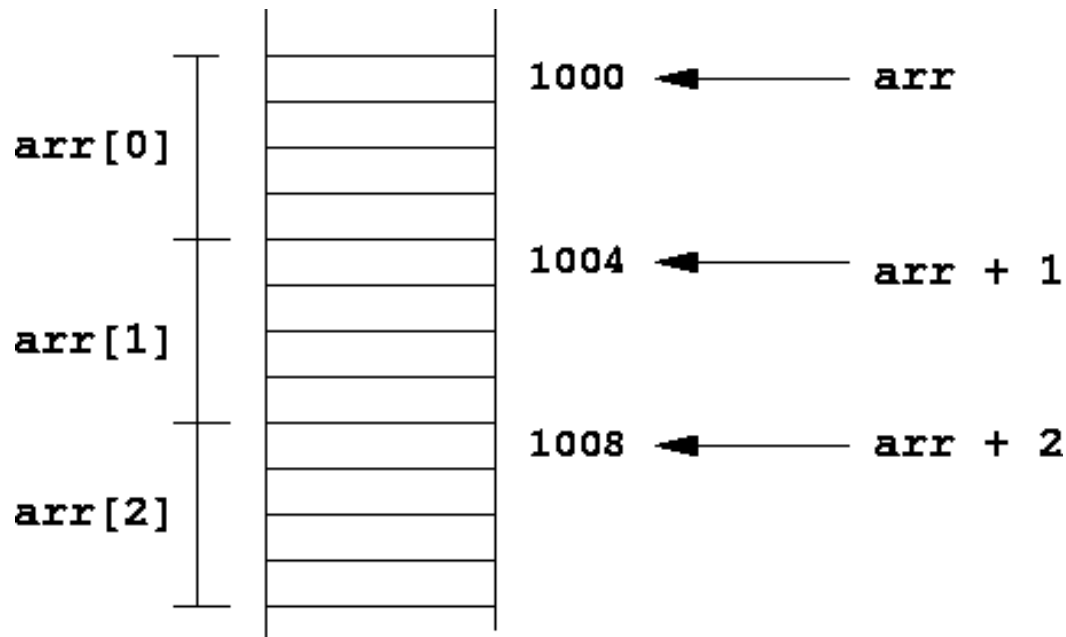
---

- Έγκυρες πράξεις με δείκτες είναι:
  - απόδοση τιμής με δείκτη ίδιου τύπου
  - η πρόσθεση ή αφαίρεση δείκτη και ακεραίου
  - η αφαίρεση ή σύγκριση δύο δεικτών για μέλη ίδιου πίνακα
  - η αντικατάσταση ή σύγκριση με NULL
  
- Δεν είναι έγκυρες πράξεις οι:
  - πρόσθεση δύο δεικτών
  - πολλαπλασιασμός, διαίρεση, ολίσθηση, η πρόσθεση float, double σε δείκτες.

# Pointer Arithmetic (Αριθμητική Δεικτών)

- Ας θεωρήσουμε ότι έχουμε ένα array ακεραίων **int arr[10]**
- Γνωρίζουμε πως **arr[i]** είναι το i-οστό στοιχείο του array
- Αυτό που δεν γνωρίζουμε είναι πως το **arr[i]** ορίζεται ως **\*(arr + i)**

```
arr == &arr[0]  
arr[0] == *arr  
arr[i] == *(arr + i)
```

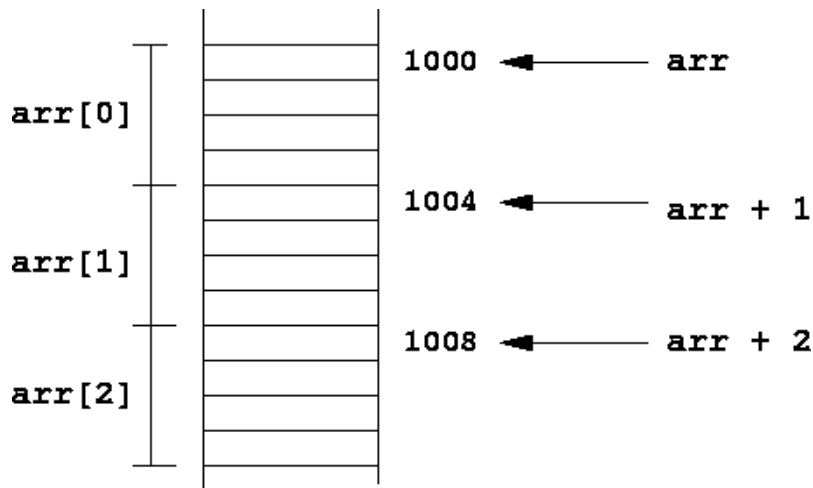


# Pointer Arithmetic (Αριθμητική Δεικτών)

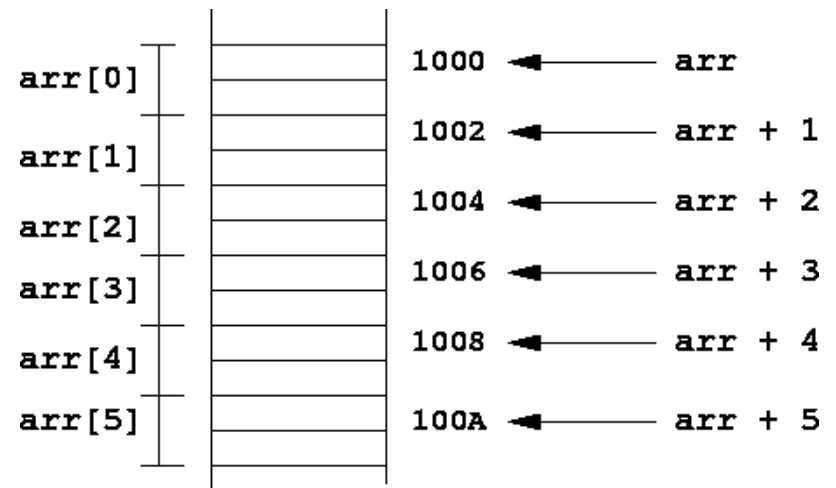
---

Το κατά πόσο αυξάνεται/μειώνεται ένας pointer σε πράξεις με ακεραίους, εξαρτάται από τον τύπο του.

**int arr[10];**  
**sizeof(int) == 4**



**short arr[10];**  
**sizeof(short) == 2**



# Δείκτες – Πίνακες (1)

---

- Το όνομα ενός πίνακα είναι μία «σταθερά τύπου δείκτη» με τιμή τη διεύθυνση του πρώτου στοιχείου του πίνακα.
- Λόγω αυτού του γεγονότος μπορούμε να ορίσουμε ένα δείκτη στον πίνακα και να χειριστούμε τον πίνακα μέσω αυτού του δείκτη.
- Δεν είναι εφικτή η ανάθεση τιμής και η αριθμητική με δείκτες

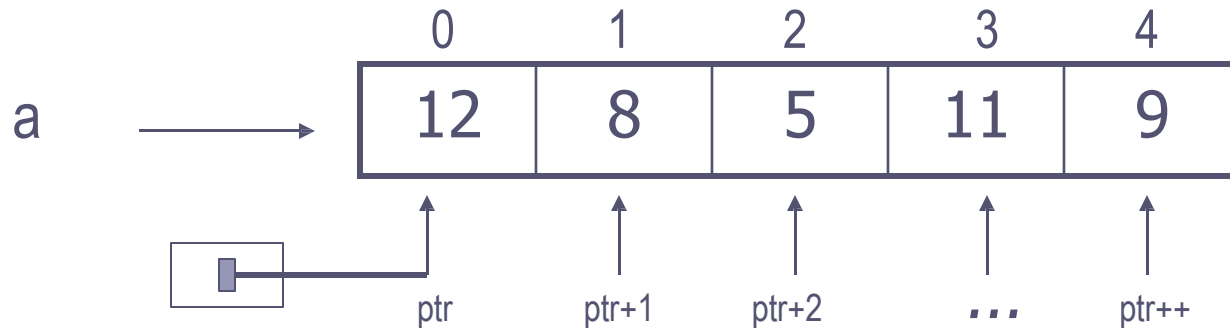
# Δείκτες – Πίνακες (2)

Έστω

```
int a[5] = { 12, 8, 5, 11, 9};
```

```
int *ptr;
```

```
ptr = a; ή ptr = &a[0];
```

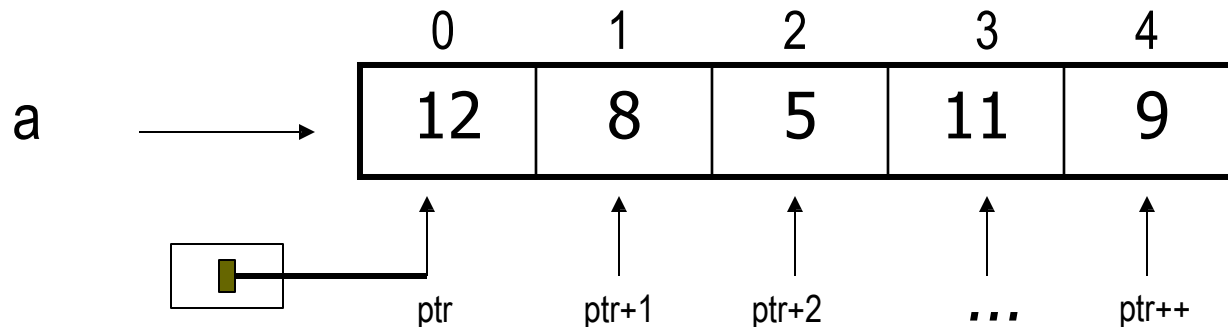


Προσοχή: `a++`; `a--` (το όνομα του πίνακα δεν είναι δείκτης (δηλ. μεταβλητή δείκτη))

# Δείκτες – Πίνακες (3)

- Μπορούμε να δώσουμε σε έναν pointer την διεύθυνση ενός array, και στη συνέχεια να τον χρησιμοποιήσουμε ως array

```
int a[] = {12, 8, 5, 11, 9}, *ptr;  
ptr=a;  
printf("%d\n", ptr[1]);    // 8  
ptr += 2;  
printf("%d\n", ptr-a);    // 2  
printf("%d\n", ptr[1]);    // 11  
ptr--;  
printf("%d\n", ptr[1]);    // 5  
a++; // Compile error, arr is const!
```



# Πέρασμα Παραμέτρων

---

- Κατ' αξία ή τιμή (by value)

Η συνάρτηση δουλεύει σε αντίγραφα των πραγματικών παραμέτρων

- Κατ' αναφορά (by reference)

Η συνάρτηση δουλεύει στις πραγματικές παραμέτρους (μόνο για πίνακες)

# Πέρασμα Μεταβλητών

---

```
#include <stdio.h>
void swap(int x, int y);

int main()
{
    int x=2, y=3;
    printf("x:%d, y:%d \n", x,y);
    swap(&x,&y);
    printf("x:%d, y:%d \n", x,y);
}

void swap(int x, int y)  /* λάθος */
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
```



# Πέρασμα Μεταβλητών (2)

---

```
#include <stdio.h>
void swap(int *px, int *py);

int main()
{
    int x=2, y=3;
    printf("x:%d, y:%d \n", x,y);
    swap(&x,&y);
    printf("x:%d, y:%d \n", x,y);
}

void swap(int *px, int *py)    /*σωστή υλοποίηση
*/
{
    int temp;
    temp=*px;
    *px=*py;
    *py=temp;
}
```

# Δυναμική Διαχείριση Μνήμης

---

- ***Calloc, Malloc,***  
για δέσμευση μνήμης
- ***Realloc***  
για αλλαγή μεγέθους δεσμευμένης μνήμης
- ***Free***  
για απελευθέρωση μνήμης

# Malloc()

---

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

```
void *malloc(size_t size);
```

Η συνάρτηση malloc δεσμεύει χώρο μεγέθους τουλάχιστον size bytes.

Επιστρέφει ένα δείκτη στη δεσμευθείσα μνήμη ή NULL, αν δεν υπάρχει διαθέσιμη μνήμη. Τα bytes του χώρου μνήμης δεν παίρνουν αρχική τιμή

```
p=(int *)malloc(n*sizeof(int)); // int *p;
```

# Calloc()

---

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

```
void *calloc(size_t n, size_t size);
```

Η συνάρτηση calloc δεσμεύει χώρο για ένα πίνακα n στοιχείων, μεγέθους size το καθένα. Διασφαλίζει την αρχικοποίησή της με 0.

Επιστρέφει ένα δείκτη στη δεσμευθείσα μνήμη ή NULL, αν δεν υπάρχει διαθέσιμη μνήμη.

```
p=(int *)calloc(n, sizeof(int));
```

# Realloc()

---

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

```
void *realloc(void *buffer, size_t size);
```

Η συνάρτηση `realloc` μεταβάλλει το μέγεθος ενός τμήματος μνήμης που είχε προηγουμένα δεσμευτεί. Το νέο μέγεθος ορίζεται από τη `size`. Διασφαλίζει τα υπάρχοντα περιεχόμενα στη μνήμη. Επιστρέφει ένα δείκτη στο νέο τμήμα μνήμης που μπορεί να είναι ίδιος με τον `buffer` ή διαφορετικός αν το τμήμα μετακινηθεί. Αν ο `buffer` είναι `NULL`, η `realloc` λειτουργεί σαν τη `malloc`.

```
p=(int *) realloc(p, n*sizeof(int));
```

# Free ()

---

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

```
void free(void * buffer);
```

Η συνάρτηση `free` αποδεσμεύει τη μνήμη η οποία σηματοδοτείται από το όρισμα `buffer` και η οποία είχε προηγουμένα δεσμευτεί με κλήση μίας εκ των *calloc*, *malloc*, *realloc*.