

## Δείκτες – pointers

Όταν δηλώνεται μια μεταβλητή, `float x;` ο compiler κρατά μια θέση στην μνήμη

Κάθε byte μνήμης έχει συγκεκριμένη διεύθυνση

Γνώση της διεύθυνσης βοηθά στην ταχύτερη εκτέλεση του προγράμματος

Η διεύθυνση στη μνήμη που είναι αποθηκευμένη η μεταβλητή `x` βρίσκεται γράφοντας:

`&x`  διεύθυνση μνήμης της `x`

Μπορούμε να δηλώσουμε μεταβλητές που περιέχουν διευθύνσεις μνήμης

Οι μεταβλητές αυτές ονομάζονται **pointers**

Μια μεταβλητή, `xp`, τύπου `pointer` ορίζεται δηλώνοντας τον τύπο της μεταβλητής που είναι αποθηκευμένη στην διεύθυνση μνήμης ακολουθούμενη από ένα \*

`float *xp;`  δήλωση μεταβλητής τύπου `pointer`

Οι μεταβλητές τύπου `pointer` παίρνουν μόνο τιμές διευθύνσεων.

`float *xp = &x;`  δήλωση μεταβλητής `pointer` σε διεύθυνση μνήμης που κρατά `float` τιμή και ανάθεση της διεύθυνσης μίας `float` μεταβλητής


- Ο τύπος του `pointer` δεν μπορεί να είναι διαφορετικός από τον τύπο της τιμής που είναι αποθηκευμένη στην διεύθυνση μνήμης που δείχνει ο `pointer`

`double x; float *xp;`

`xp = &x;`  Λάθος ανάθεση – διαφορετικοί τύποι

## Pointers – γενικά

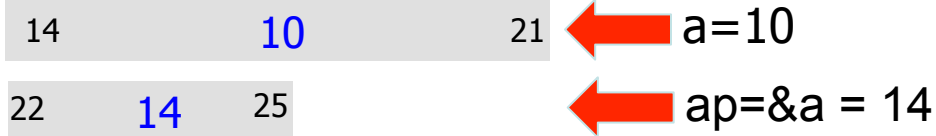
Η δήλωση μιας μεταβλητής προδιαθέτει μια σειρά από bytes της μνήμης του Η/Υ.

**double a;**  8 bytes    έστω βρίσκονται στις θέσεις 14-21  
 όταν πούμε α=10; τότε το 10 αποθηκεύεται στα bytes 14-21  
 η διεύθυνση της μνήμης δίνεται από τη διεύθυνση του 1<sup>ου</sup> byte - 14

Έστω δηλώνουμε έναν pointer. **double \*ap;**

Η παραπάνω δήλωση για υπολογιστή 32-bits σημαίνει ότι έχουν κρατηθεί 4 bytes για την ap  
 έστω οι θέσεις των bytes αυτών είναι 22-25

Έστω ότι κάνουμε ανάθεση στην ap την διεύθυνση της α: **ap = &a;**  ap=14



# Pointers

Πως μπορούμε να βρούμε την τιμή στην διεύθυνση που δείχνει ένας pointer;

**\*xp**  τιμή αποθηκευμένη στην διεύθυνση μνήμης που δείχνει ο xp

```
float y; float *xp = &x;  
y = *xp;
```

Η πράξη \* αναιρεί την πράξη του &

**y = \*(&x);**  Ισοδύναμο με y=x

Λέμε ότι η πράξη \* αφαιρεί την αναφορά (**dereference**) τον pointer

➤ Γιατί όμως είναι χρήσιμοι οι pointers?

- ☐ Τα ορίσματα των συναρτήσεων περνούν κατά τιμές και όχι κατά αναφορά.
- ☐ Οι τιμές των μεταβλητών στα ορίσματα δεν αλλάζουν μεταξύ του υποπρογράμματος που καλεί και της συνάρτησης που καλείται
- ☐ Συχνά όμως θέλουμε να περάσουμε νέες τιμές μεταξύ των δυο προγραμμάτων
- ☐ Στη FORTRAN όλα τα ορίσματα των συναρτήσεων ή των subroutines περνούν κατά αναφορά και όχι κατά τιμή. Έτσι οι τιμές αλλάζουν και τα υποπρογράμματα επικοινωνούν μεταξύ τους

# Πίνακες – arrays

Ένας πίνακας χρησιμοποιεί διαδοδικές θέσεις μνήμης για αποθήκευση τιμών

**double a[5];**  Μονοδιάστατος πίνακας 5 στοιχείων

Το 1<sup>ο</sup> στοιχείο είναι το **a[0]** και το τελευταίο το **a[4]**

Όπως είδαμε μπορούμε να δώσουμε τιμές: **double a[5] = {1., 2., 3., 4., 5.};**

Ένας πίνακας 2 διαστάσεων ορίζεται: **double b[3][2];**  3 γραμμές και 2 στήλες

14	<b>a[0]</b>	21
22	<b>a[1]</b>	29
	<b>a[2]</b>	
	<b>a[3]</b>	
	<b>a[4]</b>	
	<b>b[0][0]</b>	
	<b>b[0][1]</b>	
	<b>b[1][0]</b>	
	<b>b[1][1]</b>	
	<b>b[2][0]</b>	
	<b>b[2][1]</b>	


Παρατηρήστε τον τρόπο αποθήκευσης όπου ο 2<sup>ος</sup> δείκτης (στήλες) αλλάζει γρηγορότερα

Γράφοντας **a[3]** ή **b[1][0]** έχουμε πρόσβαση σε τιμές διπλής ακρίβειας που είναι αποθηκευμένες στις συγκεκριμένες θέσεις

➤ Γράφοντας **a** ή **b** για τους πίνακες, (χωρίς συγκεκριμένο στοιχείο) δίνουμε την διεύθυνση μνήμης του 1<sup>ου</sup> στοιχείου των πινάκων

Μπορούμε έτσι να κάνουμε ανάθεση σε pointer: **ap = a;**

Αυτό είναι ισοδύναμο με το να γράψουμε: **ap = &a[0];**

Μετά την ανάθεση αυτή μπορούμε να αλλάξουμε τη τιμή οποιουδήποτε στοιχείου του πίνακα **a**: **ap[2] = 5;**  **a[2]=5** αντί για 3.

➤ Η ανάθεση έχει μετατρέψει τον pointer έμεσα σε πίνακα

# Μεταβλητές Αναφοράς

Οι μεταβλητές αναφοράς εναλλάσσουν ονόματα για άλλη μεταβλητή

➤ Δεν δαπανούν χώρο μνήμης

Όταν δηλωθούν θα πρέπει να αποκτήσουν αρχική τιμή **double &g = a;**

Η δήλωση **double &** σημαίνει **αναφορά** σε τιμή διπλής ακρίβειας.

Είναι διαφορετικό με το & που εμφανίζεται στο δεξί μέρος μιας = που δείχνει διεύθυνση μνήμης

Αν γράψουμε **g = 5;** αυτόματα **α = 5;**

Οι μεταβλητές αναφοράς χρησιμοποιούνται όπως θα δούμε για να ανταλλάξουν τιμές ανάμεσα σε προγράμματα ή υποπρογράμματα

# Πράξεις με pointer και μεταβλητές αναφοράς

```
#include <cmath>
#include <iostream>
using namespace std;

int main()
{
    int var;
    int *pointer;

    pointer = &var;
    var = 421;
    printf("Address of the integer variable var : %p\n",&var); ← Κάποια τιμή μνήμης για την var (A)
    //
    printf("Value of var : %d\n", var); ← 421
    //
    printf("Value of the integer pointer variable: %p\n", pointer); ← η τιμή μνήμης της var (όπως στο A)
    //
    printf("Value which pointer is pointing at : %d\n", *pointer); ← 421
    //
    printf("Address of the pointer variable : %p\n", &pointer); ← διεύθυνση μνήμης της θέσης του pointer
    return 0;
}
```

# Πράξεις με pointers και μεταβλητές αναφοράς

```
#include <cmath>
#include <iostream>
int main()
{
    int matr[2];
    int *pointer;
    pointer = &matr[0];
    matr[0] = 321;
    matr[1] = 322;
```

```
    printf("\nAddress of the matrix element matr[1]: %p",&matr[0]);
```



Κάποια τιμή μνήμης για το matr[0]

```
    printf("\nValue of the matrix element  matr[1]; %d",matr[0]);
```



τιμή 321

```
    printf("\nAddress of the matrix element matr[2]: %p",&matr[1]);
```



Κάποια τιμή μνήμης για το matr[1]

```
    printf("\nValue of the matrix element  matr[2]: %d\n", matr[1]);
```



τιμή 322

```
    printf("\nValue of the pointer : %p",pointer);
```



ίδια τιμή με τη θέση μνήμης matr[0]

```
    printf("\nValue which pointer points at  : %d",*pointer);
```



τιμή 321

```
    printf("\nValue which  (pointer+1) points at: %d\n",*(pointer+1));
```



τιμή 322 δείχνει στην επόμενη θέση μνήμης, αυτή του στοιχείου matr[1]

```
    printf("\nAddress of the pointer variable: %p\n",&pointer);
```

```
    return 0;
```

```
}
```

# Πράξεις με pointers και μεταβλητές αναφοράς

```
#include <iostream>
using namespace std;
int main ()
{
    int firstvalue, secondvalue;
    int * mypointer;

    mypointer = &firstvalue;
    *mypointer = 10;
    mypointer = &secondvalue;
    *mypointer = 20;
    cout << "firstvalue is " << firstvalue << '\n';
    cout << "secondvalue is " << secondvalue << '\n';
```

```
int *newpntr1, *newpntr2;
newpntr1 = &firstvalue;
newpntr2 = &secondvalue;
```

← διεύθυνση της firstvalue  
και διεύθυνση της secondvalue

```
*newpntr1 = 5;
```

← η τιμή στη θέση που δείχνει η newpntr1 είναι 5

```
*newpntr2 = *newpntr1;
```

← η τιμή στη διεύθυνση μνήμης του newpntr2 ίδια  
με αυτή του newpntr1

```
newpntr1 = newpntr2;
```

← Η διεύθυνση του pointer2 αντιγράφεται στον pointer1

```
*newpntr1 = 15;
```

← Η νέα τιμή (15) στη θέση μνήμης pointer1

```
cout << "firstvalue is " << firstvalue << '\n';
```

← νέα τιμή firstvalue = 5

```
cout << "secondvalue is " << secondvalue << '\n';
```

← νέα τιμή secondvalue = 15

```
return 0; }
```




## Pointers – παράδειγμα

// Ορισμός μιας συνάρτησης

```
1 void f_and_df(double x, double *func, double *deriv){  
2   *func = 4*x - cos(x);  
3   *deriv = 4 + sin(x) ;  
4 }
```

Στο κύριο πρόγραμμα καλούμε την συνάρτηση με τον ακόλουθο τρόπο:

```
1 int main () {  
2   double x, f, dfdx;  
   ....  
   // κλίση της συνάρτησης  
3   f_and_df(x, &f, &dfdx);  
4   pn = po - f/dfdx;  
   ...  
5 }
```

 Το 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα είναι pointers σε μεταβλητές σε τιμές που είναι double

### Παρατηρήσεις:

- Η συνάρτηση δεν επιστρέφει τιμή, για το λόγο αυτό ορίζεται τύπου **void** (1<sup>η</sup> γραμμή)
- Στο κύριο πρόγραμμα δεν υπάρχει ανάθεση ή αλγεβρική πράξη αλλά απλά κλίση του ονόματος της συνάντησης (3<sup>η</sup> γραμμή του main)
- Το 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα της συνάρτησης είναι οι τιμές που επιστρέφει και χρησιμοποιούνται στο main. Τις περνάμε σαν pointers (1<sup>η</sup> γραμμή και 3<sup>η</sup> του main)

## Pointers – παράδειγμα

// Ορισμός μιας συνάρτησης

```
1 void f_and_df(double x, double *func, double *deriv){  
2   *func = 4*x - cos(x);  
3   *deriv = 4 + sin(x) ;  
4 }
```

Στο κύριο πρόγραμμα καλούμε την συνάρτηση με τον ακόλουθο τρόπο:

```
1 int main () {  
2   double x, f, dfdx;  
   ....  
   // κλίση της συνάρτησης  
3   f_and_df(x, &f, &dfdx);  
4   pn = po - f/dfdx;  
   ...  
5 }
```



Το 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα είναι pointers σε μεταβλητές σε τιμές που είναι double

### Παρατηρήσεις:

- Στο main ορίζουμε τις μεταβλητές f, dfdx που θα λάβουν τις τιμές που επιστρέφει η function. Καλούμε την συνάρτηση βάζοντας στο 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα τις διευθύνσεις των μεταβλητών (άρα pointers). Επομένως θα πρέπει ο ορισμός της συνάρτησης να χρησιμοποιεί 2 pointers σαν ορίσματα, όπως και συμβαίνει θέτοντας double\*func και double\*deriv που είναι του ίδιου τύπου (double) με τις μεταβλητές f και dfdx

## Pointers – παράδειγμα

// Ορισμός μιας συνάρτησης

```
1 void f_and_df(double x, double *func, double *deriv){  
2   *func = 4*x - cos(x);  
3   *deriv = 4 + sin(x);  
4 }
```

← \*func δηλώνει ανάθεση τιμής στη διεύθυνση μνήμης func. Η διεύθυνση είναι ακέραιος

Στο κύριο πρόγραμμα καλούμε την συνάρτηση με τον ακόλουθο τρόπο:

```
1 int main () {  
2   double x, f, dfdx;  
   ....  
   // κλίση της συνάρτησης  
3   f_and_df(x, &f, &dfdx);  
4   pn = po - f/dfdx;  
   ...  
5 }
```


← Το 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα είναι pointers σε μεταβλητές σε τιμές που είναι double

### Παρατηρήσεις:

- Στις γραμμές 2 και 3 της συνάρτησης κάνουμε ανάθεση του αποτελέσματος των πράξεων στη θέση της μνήμης που δείχνουν οι pointers func και deriv. Το \* μπροστά από τις pointer μεταβλητές δίνει τιμή και όχι διεύθυνση. Θα ήταν λάθος να δώσουμε π.χ.:  $func = 4x - \cos(x)$  γιατί τότε θα προσπαθούσαμε να αλλάξουμε την τιμή της διεύθυνσης μνήμης και ο compiler θα δώσει λάθος.

## Ορίσματα συναρτήσεων σαν αναφορές

```
// Ορισμός μιας συνάρτησης
1 void f_and_df(double x, double &func, double &deriv){
2   func = 4*x - cos(x);
3   deriv = 4 + sin(x);
4 }
```



αναφορές

Στο κύριο πρόγραμμα καλούμε την συνάρτηση με τον ακόλουθο τρόπο:

```
1 int main () {
2   double x, f, dfdx;
3   ....
4   // κλίση της συνάρτησης
5   f_and_df(x, f, dfdx);
6   pn = po - f/dfdx;
7   ...
8 }
```

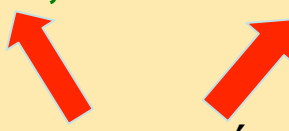
← Το 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα είναι μεταβλητές

### Παρατηρήσεις:

- Δίνοντας στον ορισμό της συνάρτησης τα ορίσματα με **&** δηλώνουμε ότι είναι αναφορές σε μεταβλητές τύπου double. Ορίζουμε έτσι ένα συμβολικό όνομα (*alias*) για το ίδιο πράγμα. Τα ορίσματα που δίνονται κατά την κλίση είναι συμβατά με τον ορισμό της συνάρτησης και δεν χρειάζεται να βρούμε τις τιμές των διευθύνσεων. Τα *func* και *deriv* είναι απλά *aliases* των *f* και *dfdx*

## Ορίσματα συναρτήσεων σαν αναφορές

```
// Ορισμός μιας συνάρτησης
1 void f_and_df(double x, double &func, double &deriv){
2   func = 4*x - cos(x);
3   deriv = 4 + sin(x);
4 }
```



αναφορές

Στο κύριο πρόγραμμα καλούμε την συνάρτηση με τον ακόλουθο τρόπο:

```
1 int main () {
2   double x, f, dfdx;
3   ....
4   // κλίση της συνάρτησης
5   f_and_df(x, f, dfdx);
6   pn = po - f/dfdx;
7   ...
8 }
```

← Το 2<sup>ο</sup> και 3<sup>ο</sup> όρισμα είναι μεταβλητές

### Παρατηρήσεις:

- Αν δεν ξέραμε τον τρόπο κατασκευής της συνάρτησης τότε η κλίση με όρισμα f, dfdx θα ήταν σα να περνούσαμε τις τιμές μόνο. Θα καταλαβαίναμε αν πρόκειται για όρισμα alias αν προσπαθούσαμε να περάσουμε σαν όρισμα το αποτέλεσμα κάποιας αλγεβρικής πράξης ενώ αναμένονταν αναφορά π.χ. `f_and_df(x, 2*f, dfdx+1)`. Αυτό θα οδηγούσε σε σφάλμα.

# Pointers και μεταβλητές αναφοράς

Στο πρόγραμμα της εύρεσης ρίζας με την μέθοδο Newton είχαμε καλέσει την function από το main με τη μορφή: `f_and_df(xp, &f, &df);`

Ενώ η συνάρτηση είχε οριστεί με την μορφή: `void f_and_df(double x, double *func, double *deriv);`

Με την κλίση της συνάρτησης αυτό το οποίο συμβαίνει είναι:

`x = xp` το x στην συνάρτηση έχει την τιμή της xp του κυρίου προγράμματος

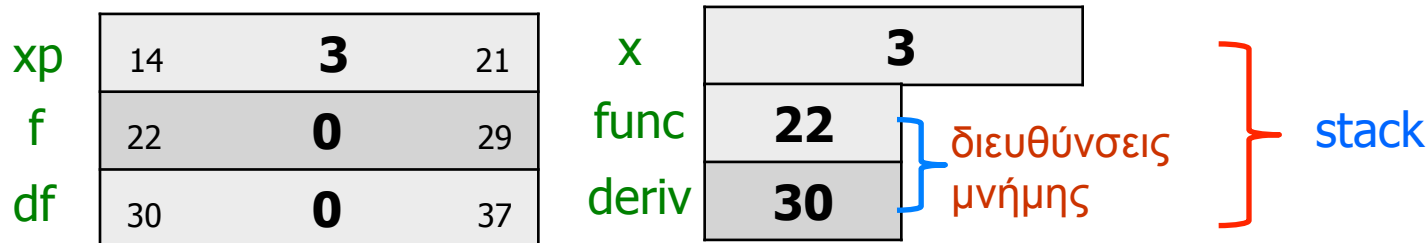
`*func = &f` η διεύθυνση μνήμης της μεταβλητής f του main περνά στην pointer μεταβλητή func

`*deriv = &df` η διεύθυνση μνήμης της df του main περνά στην pointer μεταβλητή deriv

Ο τρόπος αυτός είναι διαφορετικός από την συνηθισμένη κλίση συνάρτησης με ορίσματα.

Οι μεταβλητές που χρησιμοποιούνται σαν ορίσματα στις συναρτήσεις δεν έχουν μόνιμο χώρο στη μνήμη αλλά όταν καλείται η συνάρτηση ένας προσωρινός χώρος δημιουργείται για αποθήκευση. Ο χώρος αυτός ονομάζεται “stack”

Ας υποθέσουμε τον παρακάτω τρόπο διαχείρισης της μνήμης για την περίπτωση μας και ότι `xp=3`.



Όταν η συνάρτηση επιστρέφει τότε το stack καταστρέφεται και οι τιμές χάνονται

Αν στην συνάρτηση έχουμε `x = 10;` τότε η τιμή του x στο stack θα αλλάξει αλλά θα χαθεί επιστρέφοντας. Έτσι η τιμή του xp στο κύριο πρόγραμμα παραμένει αμετάβλητη

## Κανόνας Horner για πολυώνυμα

Ένα γενικό πολυώνυμο βαθμού  $n$  είναι:  $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n = \sum_{i=0}^n a_i x^i$

Είναι σημαντικό να γράφουμε αποδοτικό και γρήγορο κώδικα για διάφορους υπολογισμούς

Έστω ότι έχουμε το πολυώνυμο:  $7x^3 + 5x^2 - 4x + 2$

Ο εύκολος τρόπος είναι να υπολογίσουμε τα μονώνυμα  $a_i \cdot x^i$  και να τα προσθέσουμε

Αυτό όμως θα απαιτούσε για την περίπτωση μας 9 πράξεις γιατί θα είχαμε 3 πολ/μους για τον 1<sup>ο</sup> όρο, 2 πολ/σμούς για τον 2<sup>ο</sup> όρο και 1 πολ/σμό για τον 3<sup>ο</sup> όρο και 3 προσθέσεις

Εν γένει απαιτούνται  $n(n+3)/2$  πράξεις για κάθε πολύωνυμο  $n$ -βαθμού

Ένας καλύτερος τρόπος είναι να κρατάμε τον προηγούμενο όρο του μονωνύμου και να τον χρησιμοποιούμε για τον επόμενο. Στο παράδειγμα αν είχαμε το  $x^2$  θα μπορούσαμε με 1 πολ/σμό να βρούμε το  $x^3$ .

Αυτό ελαττώνει τον αριθμό των πράξεων από  $n(n+3)/2$  σε  $3n-1$

Ιδιαίτερα σημαντικό για πολυώνυμα μεγάλου βαθμού

Ένας ακόμα καλύτερος τρόπος είναι να ομαδοποιήσουμε το πολυώνυμο στην μορφή:

$$7x^3 + 5x^2 - 4x + 2 = 2 + x[-4 + x(5 + 7x)]$$


το οποίο μπορεί να γενικευτεί για οποιοδήποτε πολυώνυμο

Ξεκινούμε από τα δύο μονώνυμα υψηλότερης τάξης συνεχίζουμε προς τα έξω, πολ/ζοντας κάθε προηγούμενο αποτέλεσμα με  $x$ , προσθέτοντας τον όρο του μονωνύμου της χαμηλότερης τάξης

# Κανόνες Horner – πίνακες σαν ορίσματα

```

1  #include <iostream>
2  #define MAX 50;
3  using namespace std;
4  void poly(int n, double a[], const double x0, double &p) {
5      int i;
6      p = a[n];
7      for (i=n-1; i >=0 ; i--){
8          p = a[i] + x0*p;
9      }
10 }
```


**δήλωση ορίσματος που αναφέρεται σε 1-Δ πίνακα**  
**Οι κενές αγκύλες [] δηλώνουν ότι a είναι pointer σε double**  
**ΔΕΝ είναι απαραίτητο να προσδιορίσουμε τον αριθμό των στοιχείων του πίνακα στη συνάρτηση γιατί δεν κρατιέται μνήμη στην συνάρτηση αλλά μόνο στο κύριο πρόγραμμα.**

**Η σύνταξη είναι ισοδύναμη με:**

void poly(int n, double \*a, const double x0, double &p)

```

11 int main (){
12     int i, n;
13     double x0, p;
14     double a[MAX+1];
15     cout<< "Degree polynomial =";
16     cin >> n;
17     if (n>MAX) {
18         cerr << "Degree must be less than 50" << endl;
19         return 1;
20     }
21     cout << "coefficients, a[0],a[1],...,a["<<n<<"]\n";
22     for (i=0; i<=n; i++) cin >> a[i];
23     cout << "x0 = ";
24     cin << x0;
25     poly(n,a,x0,p);
26     cout << "P(" << x0 << ") = " << p << "\n";
27     return 0;
28 }
```


**ορισμός πίνακα α μεγέθους [MAX+1].**  
**Η α είναι pointer στη θέση μνήμης του a[0]**

Πίνακες δύο διαστάσεων μπορούν να δηλωθούν και να περάσουν με τον τρόπο αυτό:

```

void func(double ab[10][5]){
    ....
}
```

```

int main(){
    double b[10][5];
    func(b);
    return 0; }
```



# Pointers as arguments

```
#include <iostream>
using namespace std;

void increment_all (int *start, int *stop)
{
    int *current = start;
    while (current != stop) {
        ++(*current); // αύξηση της τιμής που είναι στη θέση μνήμης current
        ++current;    // αύξηση της τιμής της διεύθυνσης της μνήμης
    }
}

void print_all (const int *start, const int *stop)
{
    const int *current = start;
    while (current != stop) {
        cout << *current << '\n';
        ++current;    // αύξηση της τιμής της διεύθυνσης της μνήμης
    }
}

int main ()
{
    int numbers[] = {10,20,30};
    increment_all (numbers,numbers+3);
    print_all (numbers,numbers+3);
    return 0;
}
```

# Arrays σε συναρτήσεις

```
// arrays as parameters  
// For a function one could use void procedure  
// (int myarray[][3][4])
```

```
#include <iostream>  
using namespace std;
```

```
void printarray (int arg[], int length) {  
    for (int n=0; n<length; ++n)  
        cout << arg[n] << ' '  
        cout << '\n';  
}
```

```
int main ()  
{  
    int firstarray[] = {5, 10, 15};  
    int secondarray[] = {2, 4, 6, 8, 10};  
    printarray (firstarray,3);  
    printarray (secondarray,5);  
    return 0;  
}
```

# Μικρή επανάληψη από pointers

```

#include <iostream>
using namespace std;
int main() {
    int p[] = {10,20,30,40};
    int *q = p;
    int x;
    x = *q;                ← x ισούται με τη τιμή του ακεραίου που δείχνει το q
    cout << x << endl;
    x = *q++;              ← x ισούται με τη τιμή του ακεραίου που δείχνει το q (άρα 10).
                           Μετά το q δείχνει στη διεύθυνση του επόμενου ακεραίου (p[1])
    cout << x << endl;
    x = *(q++);            ← x ισούται με τη τιμή του ακεραίου που δείχνει το q (άρα 20 από το προηγούμενο).
                           Μετά το q αυξάνει και δείχνει στη διεύθυνση του επόμενου ακεραίου (p[2])
    cout << x << endl;
    x = (*q)++;            ← x ισούται με τη τιμή του ακεραίου που δείχνει το q (άρα 20 από το προηγούμενο).
                           Μετά η τιμή στη διεύθυνση του q (εδώ το στοιχείο p[2] θα έχει πλέον τιμή 31)
    cout << x << endl;
    x = *q;                ← x ισούται με τη τιμή του ακεραίου που δείχνει το q (άρα 31 από το προηγούμενο).
    cout << x << endl;
    x = *++q;              ← πρώτα αυξάνει το q κατά 1 (άρα δείχνει στη διεύθυνση του p[4]) και
                           μετά γίνεται ανάθεση στο x την τιμή στην θέση p[4] (άρα x=40).
    cout << x << endl;
    x = ++*q;              ← πρώτα αυξάνει τη τιμή της διεύθυνσης του q κατά 1 (επομένως p[4]=41) και
                           μετά κάνει ανάθεση της τιμής στο x (x=41)
    cout << x << endl;
    return 0;
}

```