

ΕΠΛ232 – Προγραμματιστικές Τεχνικές και Εργαλεία

Δείκτες και Συναρτήσεις (Κεφάλαιο 11, ΚΝΚ-2ΕΔ)

Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου

<http://www.cs.ucy.ac.cy/courses/EPL232>



Περιεχόμενο Διάλεξης 5

- **Μεταβλητές Δεικτών & Τελεστές**
 - Δήλωση και Αρχικοποίηση (NULL)
 - Τελεστής Διεύθυνσης (&), Τελεστής Έμμεσης Αναφοράς (*), Τελεστές Ανάθεσης (=), Παραδείγματα
- **Δείκτες ως Ορίσματα**
 - Δια-τιμής και Δια-Διεύθυνσης (Αναφοράς)
 - Παραδείγματα: swap, decompose, maxmin
- **Προστασία Ορισμάτων**
 - Η δήλωση `const`
 - Δείκτες ως Τιμές Επιστροφής (`void *func()`)



Μνήμη & Διευθύνσεις (Επανάληψη)

- Το πρώτο βήμα για την κατανόηση των δεικτών είναι η απεικόνιση του τι αντιπροσωπεύουν σε επίπεδο μηχανής.
 - Την μνήμη μπορούμε να την φανταστούμε ως ένα μεγάλο **πίνακα από bytes (8 bits)**, π.χ.,
 - 2 gigabytes = 2 147 483 648 Bytes
 - 4 gigabytes = 4 294 967 296 Bytes (max για 32-bit συστήματα)
- Τα bytes αυτά παραχωρούνται στις **διεργασίες** από τον **πυρήνα**.
 - Κάθε **byte** έχει μια μοναδική **διεύθυνση (address)**

0	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---



Μνήμη & Διευθύνσεις (Επανάληψη)

- Οι **διευθύνσεις** της κάθε διεργασίας (προγράμματος) είναι **ανεξάρτητες μεταξύ τους**.
 - Δηλαδή, Εάν υπάρχουν n bytes στη μνήμη κάθε διεργασία έχει διευθύνσεις μνήμης που εκτείνονται από το **0** μέχρι το **$n-1$** .
 - Λόγω της **νοητής μνήμης (virtual memory)**, ο πίνακας αυτός μπορεί να είναι όσο μεγάλη είναι η δευτερεύουσα σας μνήμη!

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
$n-1$	01000011



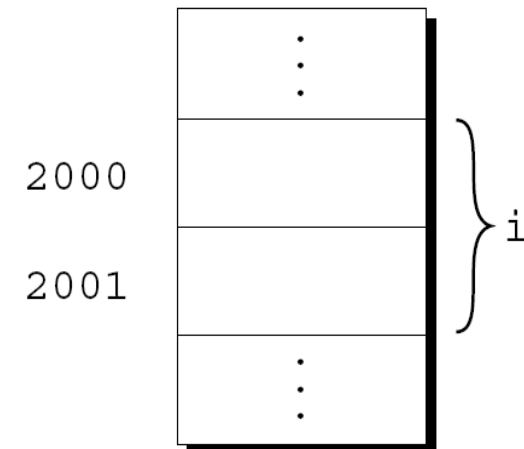
Μεταβλητή Τύπου Δείκτη

- Γνωρίζουμε ότι **κάθε μεταβλητή** σε ένα πρόγραμμα **καταλαμβάνει ένα ή περισσότερα bytes** στον πίνακα μνήμης του προγράμματος
 - Π.χ., char (1B), short(2B), int(4B), float(4B), double(8B).

- **Διεύθυνση Μεταβλητής:**

Η διεύθυνση του πρώτου byte της μεταβλητής.

- Εάν έχουμε τη δήλωση: `short i;`
- Η διεύθυνση του `i` είναι το 2000

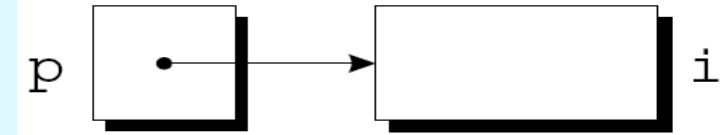


Μεταβλητή Τύπου Δείκτη

- Όταν αποθηκεύουμε τη διεύθυνση της μεταβλητής i στη μεταβλητή p , τότε λέμε ότι το p “δείχνει στο” i (ή ότι ο p είναι δείκτης στο i)

- Η γραφική απεικόνιση

- Το βελάκι δεν υπάρχει, είναι απλά για απεικόνιση!



- Οι δείκτες (p) αποτελούνται ουσιαστικά από μια ακέραια τιμή της διεύθυνσης που δείχνουν (του i).
 - Το p καταλαμβάνει χώρο: 4Bytes (ILP32) ή 8Bytes (LP64)
- Παρόλο που το p μοιάζει με μια απλή ακέραια τιμή (τύπου **unsigned int**), στην πράξη αποθηκεύονται σε ειδικές μεταβλητές **τύπου δείκτη (*pointer variables*)**.
 - Αυτό γίνεται για να επιτρέπεται η αριθμητική δεικτών που θα δούμε στη συνέχεια...



Δήλωση Μεταβλητών Δεικτών

- **Δήλωση Μεταβλητής Δείκτη (σε ακέραιο):**

```
int *p;
```

- όταν δηλώνεται μια μεταβλητή δείκτη, το όνομά της πρέπει να προηγείται ενός αστερίσκου
- `p` είναι πλέον δείκτης που μπορεί να δείχνει (point) σε αντικείμενα τύπου `int`.

- Η δήλωση μπορεί να γίνει ταυτόχρονα με άλλες:

```
int i, j, a[10], b[20], *p, *q;
```

- Η C απαιτεί ότι κάθε μεταβλητή δείκτη δείχνει μόνο σε αντικείμενα συγκεκριμένου τύπου **(του τύπου αναφοράς - *referenced type*)**:

```
int *p;      /* points only to integers    */
double *q;   /* points only to doubles      */
char *r;     /* points only to characters    */
```



Τελεστές Διεύθυνσης (&) και Έμμεσης Αναφοράς (*)

- Η C παρέχει **δύο τελεστές** οι οποίοι παρέχονται ειδικά για τη χρήση με τους δείκτες
 - Αυτό πέρα από την ανάθεση (=), ισότητας (==), κτλ.

- Οι **τελεστές** αυτοί είναι:

- Ο **Τελεστής Διεύθυνσης (&, Address Operator)**: Εύρεση της διεύθυνσης μιας μεταβλητής
 - οποιουδήποτε τύπου, ακόμη και τύπου δείκτη!
- Ο **Τελεστής Έμμεσης Αναφοράς (*, Indirection Operator)**: Για να προσπελάσουμε το αντικείμενο το οποίο αναφέρεται από ένα δείκτη.



Address Operator

Τελεστές Διεύθυνσης (&)

- Δηλώνοντας ένα δείκτη, **δεσμεύει** χώρο στη στοίβα του προγράμματος για τον **ίδιο τον δείκτη**, αλλά **ΔΕΝ** βάζει τον δείκτη να δείχνει κάπου:

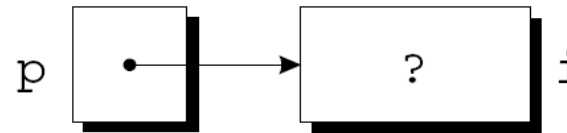
```
int *p;          /* ΕΠΙΚΙΝΔΥΝΟ: το p δείχνει οπουδήποτε*/
```

```
int *p = NULL ; /* ΑΣΦΑΛΕΣ: το p δείχνει στη διεύθυνση 0 δηλ., το NULL */
```

- Αρχικοποιείτε **ΠΑΝΤΑ** τους δείκτες σε NULL για να αποφύγετε **απρόσμενα αποτελέσματα**.
- Στη συνέχεια, μπορείτε να τους βάλετε να δείχνουν εκεί που θέλετε, π.χ.,:

```
int i, *p = NULL; p = &i;
```

```
ή int i, *p = &i;
```



- Το οποίο αποθηκεύει την διεύθυνση του **i** στη μεταβλητή **p** (δηλαδή το **p** δείχνει στο **i**) :



Indirection Operator

Τελεστής Έμμεσης Αναφοράς (*)

- **Τελεστής Έμμεσης Αναφοράς (*p):** Μας επιτρέπει να προσπελάσουμε την τιμή (περιεχόμενο) του αντικειμένου που δεικνύεται από το δείκτη p. π.χ., `printf("%d\n", *p);`
- **ΠΡΟΣΟΧΗ:** Προϋποθέτει ότι ο δείκτης δείχνει ήδη σε κάποιο αντικείμενο.
 - `int i, *p = &i; printf("%d\n", *p);` OK
 - `int *p; printf("%d\n", *p);` // ERROR
 - `int *p; *p = 1;` // ΕΠΙΚΙΝΔΥΝΟ (γράφουμε σε απροσδιόριστη διεύθυνση μνήμης)
- Το `&` μπορεί να θεωρηθεί ως η αντίστροφη πράξη του `*`, `int i, j:`
`j = *&i; /* ίδιο με j = i; */`
`j = &*i /* compile error i not pointer */`

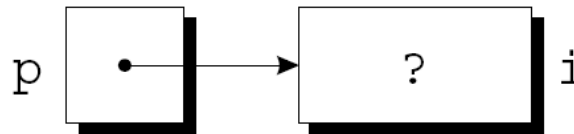


Indirection Operator

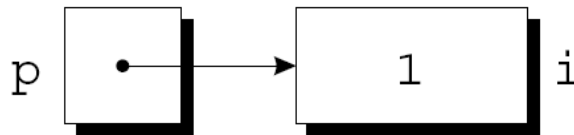
Τελεστής Έμμεσης Αναφοράς (*)

- Παράδειγμα που δείχνει ότι το `*p` και `i` αναφέρονται στην ίδια μεταβλητή

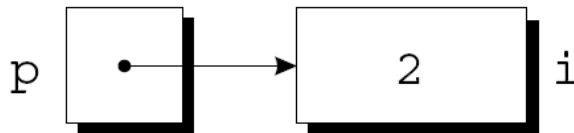
```
p = &i;
```



```
i = 1;
```



```
printf("%d\n", i); /* prints 1 */  
printf("%d\n", *p); /* prints 1 */  
*p = 2;
```



```
printf("%d\n", i); /* prints 2 */  
printf("%d\n", *p); /* prints 2 */
```

Τι θα τυπώσει το
printf("%ld", p);
(ή lld, llx, llu)
Τη διεύθυνση του ίδιου
του δείκτη!



Pointer Assignment

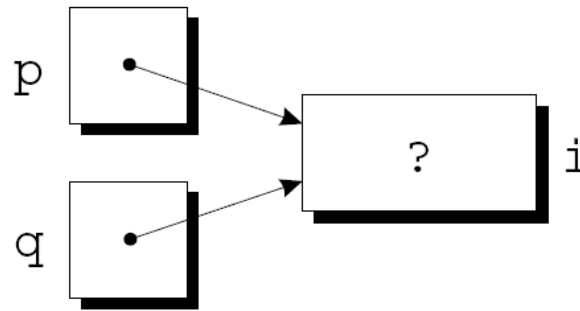
Τελεστής Ανάθεσης (=) Δεικτών

- Ο **τελεστής ανάθεσης $a = b$** , αντιγράφει το περιεχόμενο της μεταβλητής b στη a
 - Αυτό ισχύει για όλους τους βασικούς τύπους που είδαμε μέχρι στιγμής (int, float, double, κτλ.)
- Αντίστοιχα, επιτρέπεται να **αντιγράψουμε τις αναφορές των δεικτών**, αρκεί οι δείκτες να είναι του **ίδιου τύπου**.
 - Αντιγράφεται ουσιαστικά η **ακέραια διεύθυνση μνήμης** που αποθηκεύεται στην **μεταβλητή του δείκτη!**

• Π.χ.,:

```
int i, *p, *q;
```

```
p = &i; q = p;
```



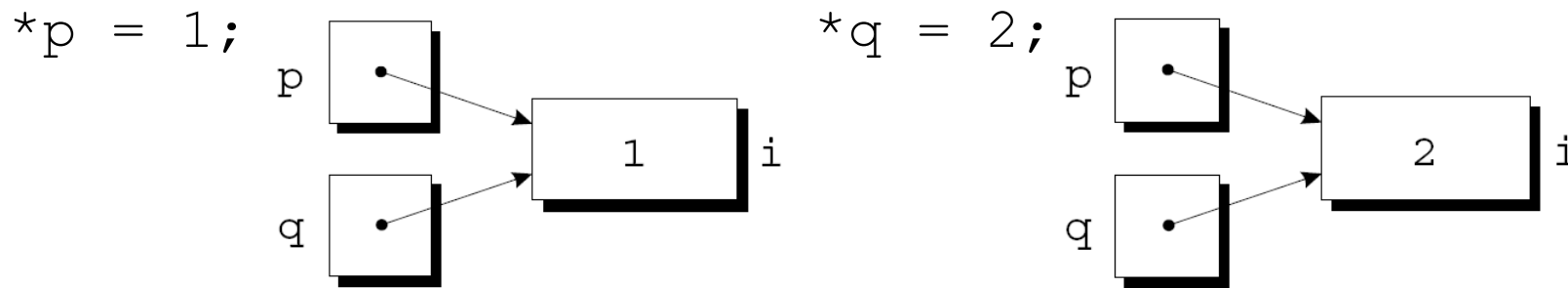
Η q τώρα δείχνει στην ίδια διεύθυνση με την p



Pointer Assignment

Τελεστής Ανάθεσης (=) Δεικτών

- Εάν p και q δείχνουν στο i , μπορούμε να αλλάξουμε το i αναθέτοντας μια νέα τιμή είτε στο $*p$ ή το $*q$:



- **Σημείωση:** Δεν υπάρχει περιορισμός στον αριθμό των δεικτών που μπορεί να δείχνουν σε ένα αντικείμενο.
- Ανάθεση Δείκτη vs. Ανάθεση Τιμής:

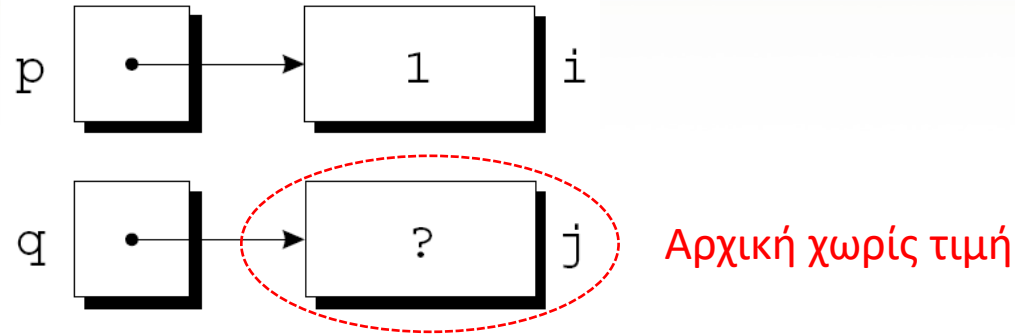
ΠΡΟΣΟΧΗ: $q = p;$ ΔΙΑΦΟΡΕΤΙΚΟ ΑΠΟ $*q = *p;$



Pointer Assignment

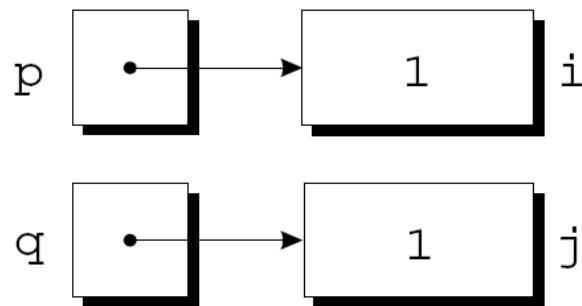
Τελεστής Ανάθεσης (=) Δεικτών

```
int i, j;  
p = &i;  
q = &j;  
i = 1;
```



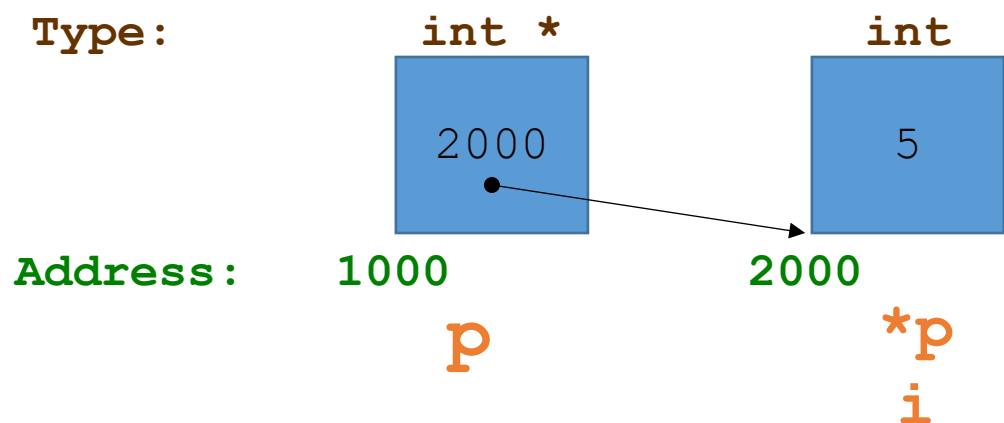
- Το περιεχόμενο του `p` δείκτη αντιγράφεται στο περιεχόμενο του `q` δείκτη

```
*q = *p;
```



Δείκτες (Pointers)

```
int *p = NULL;
int i = 5;
p = &i;
...
printf("%ld", (unsigned long int) &p); // 1000
printf("%ld", (unsigned long int) p); // 2000
printf("%d", *p); // 5
printf("%d", i); // 5
```



Παράδειγμα με Δείκτες

```
int    x = 1,    y = 2,    z[10] = {0};
int    *ip = NULL, *iq = NULL;

ip     = &x;           /* ο ip δείχνει τώρα τη διεύθυνση του x */

*ip    = *ip + 1;     /* η μεταβλητή που δείχνει ο ip (η x) αυξάνεται κατά 1 */

y      = *ip + 3;     /* η y παίρνει την τιμή 5 */
*ip    = 0;           /* η x παίρνει την τιμή 0 */

ip     = &z[6];       /* ο ip δείχνει τώρα τη διεύθυνση του z[6] */

iq     = ip;          /* ο iq δείχνει εκεί που δείχνει και ο ip */
```



Περιεχόμενο Διάλεξης 5

- **Μεταβλητές Δεικτών & Τελεστές**
 - Δήλωση και Αρχικοποίηση (NULL)
 - Τελεστής Διεύθυνσης (*), Τελεστής Έμμεσης Αναφοράς (&), Τελεστές Ανάθεσης (=), Παραδείγματα
- **Δείκτες ως Ορίσματα**
 - Δια-τιμής και Δια-Διεύθυνσης (Αναφοράς)
 - Παραδείγματα: swap, decompose, maxmin
- **Προστασία Ορισμάτων**
 - Η δήλωση `const`
 - Δείκτες ως Τιμές Επιστροφής (`void *func()`)

Pointers as Arguments

Δείκτες ως Ορίσματα

- Στη συνέχεια, θα γράψουμε τη συνάρτηση **swap** (X, Y) η οποία ανταλλάζει την τιμή της X με την τιμή της Y.

```
void swap(int *x, int *y); // ΣΩΣΤΟ
```

- Προτού δούμε την λύση, με πέρασμα μεταβλητών **δια-διεύθυνσης (by-reference)**, θα θυμίσουμε πως το πέρασμα μεταβλητών **δια-τιμής (by-value)** δεν δίνει το αναμενόμενο αποτέλεσμα.

```
void swap(int x, int y); // ΛΑΘΟΣ
```

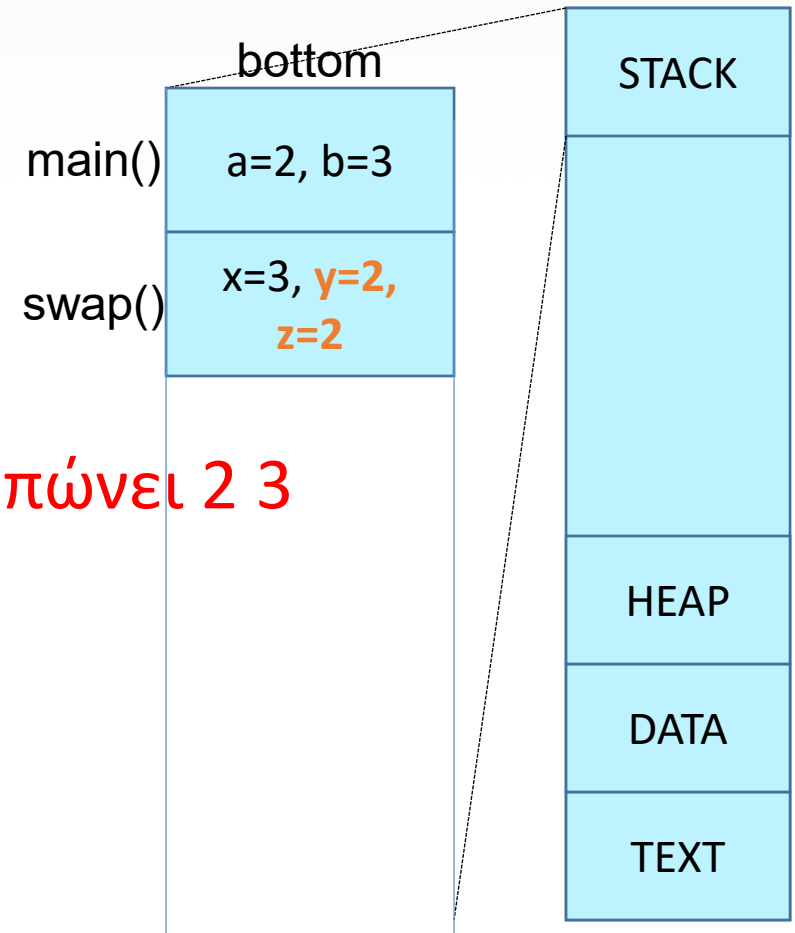


Παράδειγμα Λανθασμένη Υλοποίηση swap()

Πέρασμα-Δια-Τιμής (Pass-by-Value)

```
void swap(int x, int y) {  
    int z;  
    z = x; x = y; y = z;  
}  
  
int main() {  
    int a=2, b=3;  
    swap(a, b);  
    printf("%d %d\n", a, b);  
    return 0;  
}
```

Το πρόγραμμα τυπώνει 2 3
αντί 3 2. Γιατί;



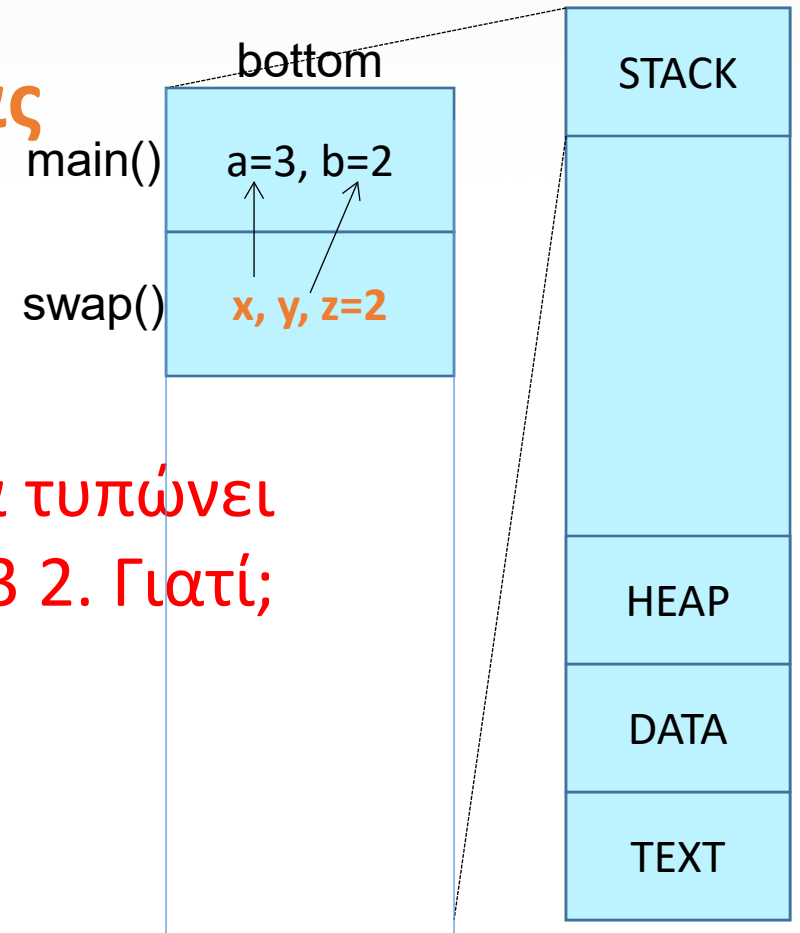
Παράδειγμα Ορθή Υλοποίηση swap()

Πέρασμα-Δια-Διεύθυνσης ή Δια-Αναφοράς (Pass-by-Address or Reference)

```
void swap(int *x, int *y) {  
    int z;  
    z = *x; *x = *y; *y = z;  
}
```

```
int main() {  
    int a=2, b=3;  
    swap(&a, &b);  
    printf("%d %d\n", a, b);  
    return 0;  
}
```

Το πρόγραμμα τώρα τυπώνει
ορθά 3 2. Γιατί;



Pointers as Arguments

Δείκτες ως Ορίσματα

- Ως δεύτερο παράδειγμα, θα γράψουμε τη συνάρτηση `decompose`, η οποία αποσυνθέτει ένα πραγματικό αριθμό `x` σε **ακέραιο όρο** `int_part` και **δεκαδική ακρίβεια** `frac_part`.
 - Η `decompose` λαμβάνει ορίσματα ως **δείκτες σε μεταβλητές**, αντί την τιμή της μεταβλητής.

```
void decompose(double x, long *int_part,  
               double *frac_part)    {  
    *int_part = (long) x;  
    *frac_part = x - *int_part;  
}
```

- Ενδεχόμενα πρότυπα συναρτήσεων της `decompose`:

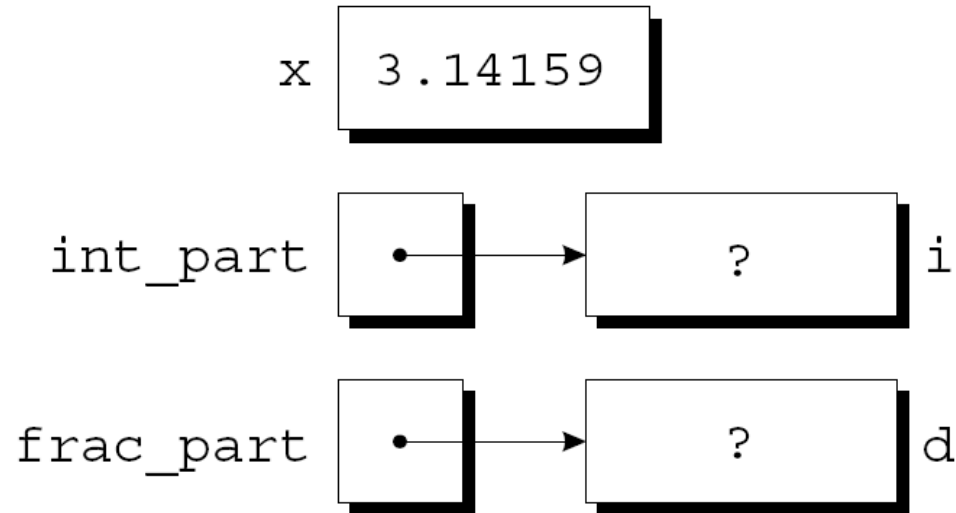
```
void decompose(double x, long *int_part,  
               double *frac_part);  
  
void decompose(double, long *, double *);
```



Pointers as Arguments

Δείκτες ως Ορίσματα

- Κλήση της `decompose`: `// long i, double d;`
`decompose(3.14159, &i, &d);`
- Ως αποτέλεσμα, το `int_part` δείχνει στο `i` και `frac_part` δείχνει στο `d` τα οποία βρίσκονται στο `scope` της καλούσας συνάρτησης:

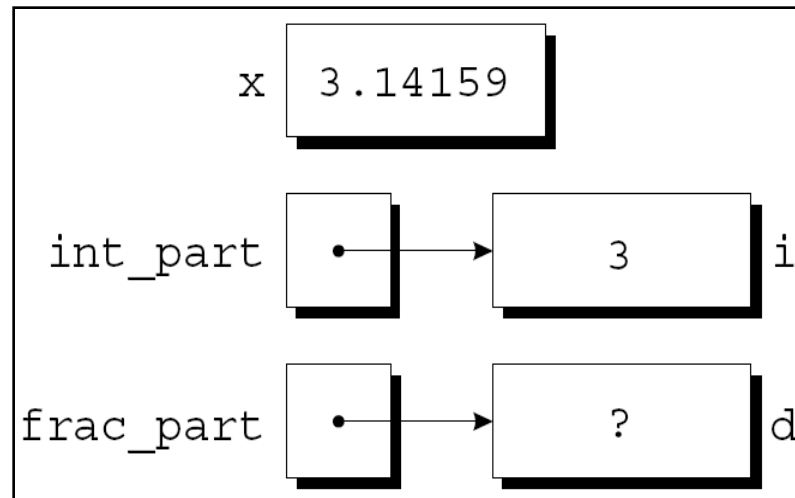


Pointers as Arguments

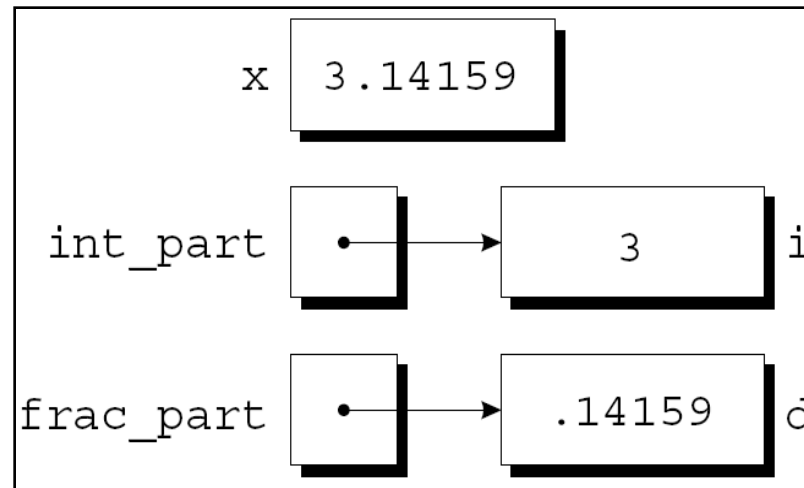
Δείκτες ως Ορίσματα

- A) Μετατρέπει την τιμή του x σε `long` και αποθηκεύει το αποτέλεσμα στο `int_part`:
- B) Αποθηκεύει το $x - *int_part$ στο αντικείμενο που δεικνύεται από το `frac_part`:

A) `*int_part = (long) x;`



B) `*frac_part = x - *int_part;`



Pointers as Arguments

Δείκτες ως Ορίσματα

- Θυμηθείτε ότι η `scanf` λάμβανε και αυτή το όρισμα της δια αναφοράς:

```
int i; scanf("%d", &i);
```

Σημείωση: Χωρίς το &, η `scanf` θα προμηθευόταν την τιμή της `i`.

ή με χρήση δείκτη

```
int i, *p; ... p = &i; scanf("%d", p);
```

- Περνώντας δια αναφορά τη διεύθυνση του δείκτη δεν θα έδινε το αναμενόμενο αποτέλεσμα

- Αυτό διότι ο δείκτης αναπαριστά ήδη διεύθυνση:

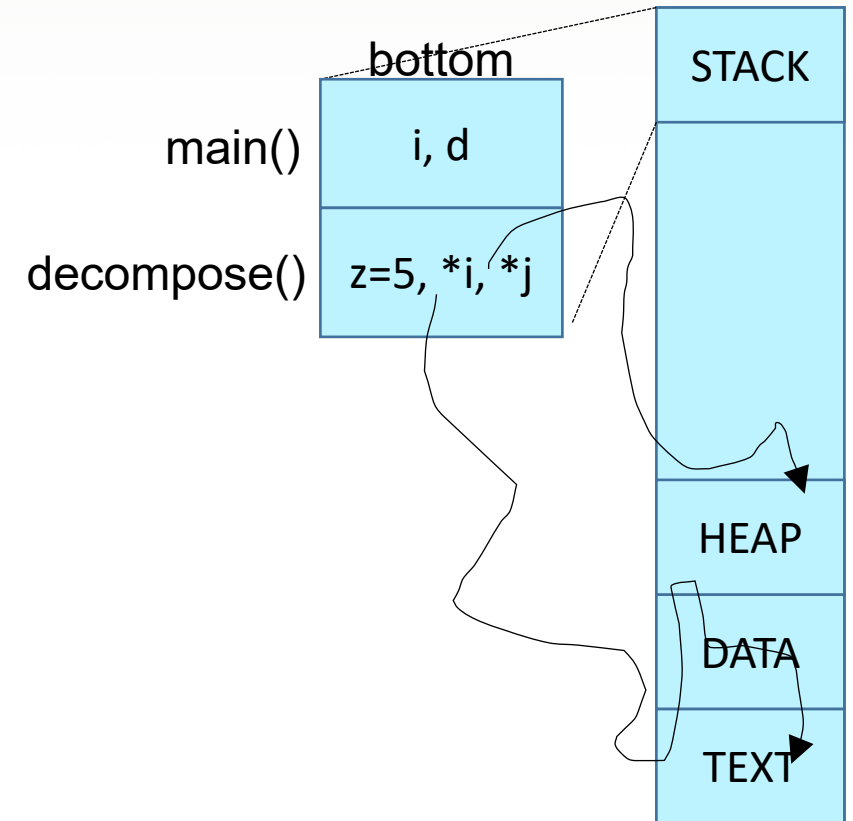
```
scanf("%d", &p); /** WRONG **/
```



Pointers as Arguments

Δείκτες ως Ορίσματα

- Η αποτυχία να περαστεί ένας δείκτης σε μια συνάρτηση, όταν είναι αναμενόμενο, μπορεί να έχει **Καταστροφικό Αποτέλεσμα**
`decompose(3.14159, i, d);`
`// long i, double d;`
- Η `decompose` επιχειρεί να αποθηκεύσει τιμές σε απροσδιόριστες περιοχές μνήμης, αντί του `i, d` στη στοίβα της καλούσας συνάρτησης, με καταστροφικό αποτέλεσμα
 - Π.χ., Segmentation Fault
- Εάν παρείχαμε πρότυπο συνάρτησης για την `decompose`, τότε ο μεταγλωττιστής θα έβρισκε το λάθος. **χωρίς &**
- Στη περίπτωση του `scanf("%d", i)` ωστόσο, θα μεταγλωττίζονταν με προειδοποίηση, άρα θα οδηγούσε σε ενδεχόμενη καταστροφή!



Παράδειγμα

MaxMin με Μεταβλητές Δια Αναφοράς

- Υλοποιήστε το ακόλουθο πρότυπο συνάρτησης `max_min.c` το οποίο βρίσκει τον **μέγιστο** και **μικρότερο** σε μια λίστα ακεραίων αποθηκεύοντας τις τιμές στο **`*max`** και **`*min`** αντίστοιχα.

```
int max_min(int a[], int n, int *max, int *min);
```

- Παράδειγμα Κλήσης της `max_min`:

```
max_min(b, N, &big, &small);
```

- Υποδειγματική Εκτέλεση:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
```

```
Largest: 102
```

```
Smallest: 7
```



Παράδειγμα

MaxMin με Μεταβλητές Δια Αναφοράς

```
/* Finds max and min elements in an array */
#include <stdio.h>
#define N 10
int max_min(int a[], int n, int *max, int *min);

int main(void)
{
    int b[N], i, big, small;
    printf("Enter %d numbers: ", N);
    for (i = 0; i < N; i++) {
        scanf("%d", &b[i]);
    }
    if ( max_min(b, N, &big, &small) == EXIT_SUCCESS );
        printf("Largest: %d\n", big);
        printf("Smallest: %d\n", small);
    } else {
        printf("Unable to execute max_min");
    }
    return 0;
}
```



Παράδειγμα

MaxMin με Μεταβλητές Δια Αναφοράς

```
int max_min(int a[], int n, int *max, int *min)
{
    int i;
    if (n<1) return (EXIT_FAILURE);

    *max = *min = a[0];
    for (i = 1; i < n; i++) {
        if (a[i] > *max)
            *max = a[i];
        else if (a[i] < *min)
            *min = a[i];
    }
    return (EXIT_SUCCESS);
}
```



Περιεχόμενο Διάλεξης

- **Μεταβλητές Δεικτών & Τελεστές**

- Δήλωση και Αρχικοποίηση (NULL)
- Τελεστής Διεύθυνσης (*), Τελεστής Έμμεσης Αναφοράς (&), Τελεστές Ανάθεσης (=), Παραδείγματα

- **Δείκτες ως Ορίσματα**

- Δια-τιμής και Δια-Διεύθυνσης (Αναφοράς)
- Παραδείγματα: `swap`, `decompose`, `maxmin`

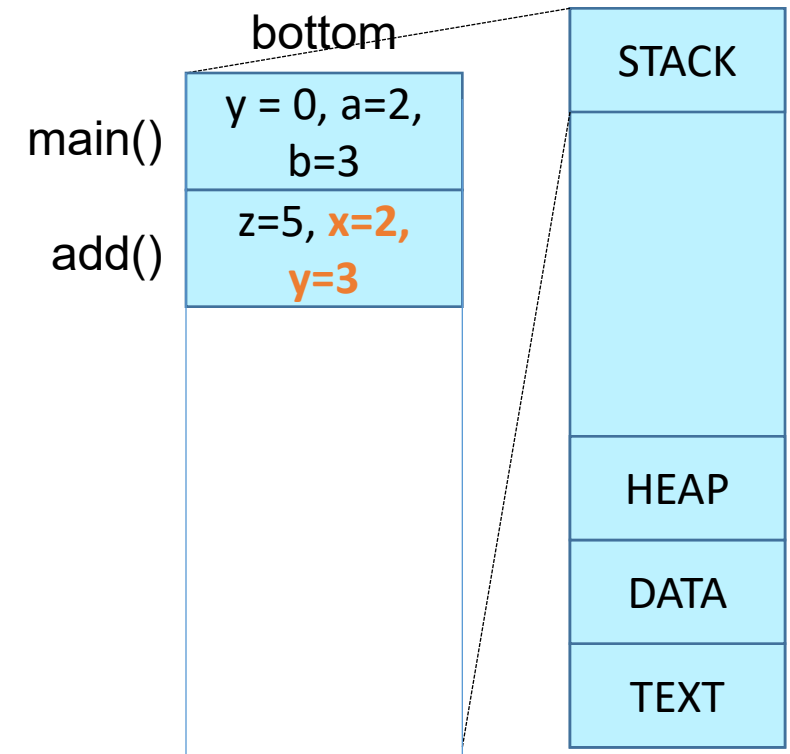
- **Προστασία Ορισμάτων**

- Η δήλωση `const`
- Δείκτες ως Τιμές Επιστροφής (`void *func()`)



Προστασία Ορισμάτων Συναρτήσεων με `const`

- Όταν μια μεταβλητή **περνά-δια-τιμής** σε μια συνάρτηση τότε **δημιουργείται ένα αντίγραφο** της στη στοίβα του προγράμματος
- `int add(int x, int y)`
- Συνεπώς, **δεν υπάρχει** η έννοια να **προστατέψουμε** μια μεταβλητή **από λανθασμένη αλλαγή** μέσα σε μια συνάρτηση.
 - Εφόσον ποτέ δεν αλλάζει η αρχική τιμή αλλά η συνάρτηση δουλεύει πάνω σε ένα αντίτυπο της

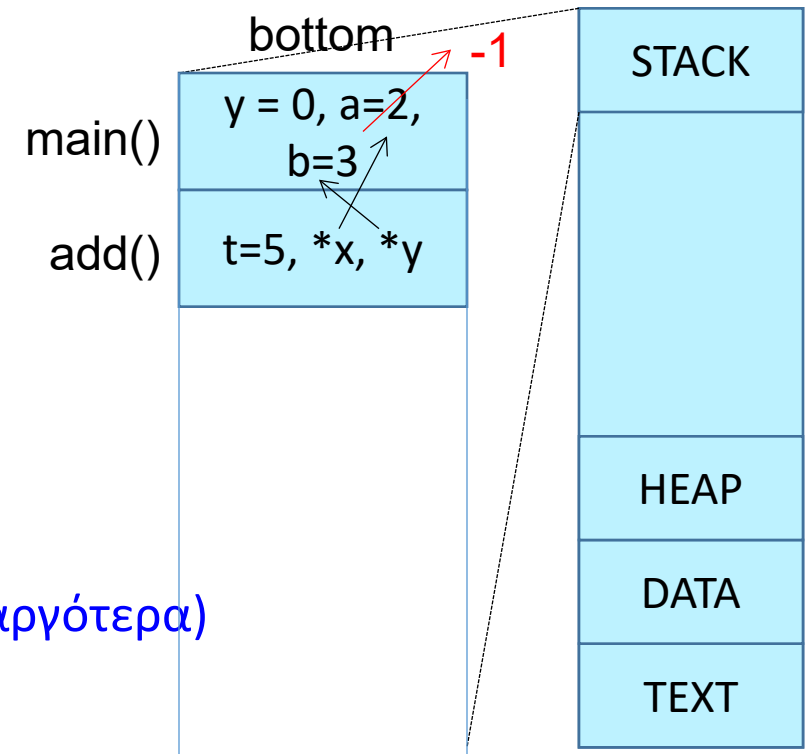


Προστασία Ορισμάτων Συναρτήσεων με `const`

- Από την άλλη, όταν η μεταβλητή **X** περνά **δια-αναφοράς** τότε **υπάρχει ο κίνδυνος** η συνάρτηση να αλλάξει το ***X** ενώ πρόθεση μας θα ήταν ίσως να διαβάσει μόνο την τιμή του ***X**., π.χ.,

```
int add(int *x, int *y);  
    int t = *x + *y;  
    *x = -1; // oups..  
    return t;  
}
```

- Τότε, γιατί να περνώ δείκτη;
 - Για λόγους επίδοσης εφόσον το αντικείμενο που περνά μπορεί να είναι πολύ μεγάλο (π.χ., struct που θα δούμε αργότερα)



Προστασία Ορισμάτων Συναρτήσεων με `const`

- **Const**: Δηλώνουμε ότι η συνάρτηση **δεν πρέπει να αλλάξει το αντικείμενο του οποίου η διεύθυνση περνά ως όρισμα στη συνάρτηση.**

Προστασία του περιεχομένου του δείκτη `p`:

```
void f(const int *p) { // Κατάλληλο για προστασία του Καλών  
    *p = 0;    /* WRONG! - Compile Error */ } Προσπάθεια τροποποίησης του *p
```

Προστασία του copy variable `p`:

```
void f(int * const p) { // Κατάλληλο για «αυτοπροστασία»  
    *p = 0;    /* OK */  
    p = 1;    /* WRONG! - Compile Error - see later **p */ }
```

Προστασία του copy variable `p` και του περιεχομένου του δείκτη:

```
void f(const int * const p) { // Κατάλληλο και για τα 2  
    *p = 0;    /* WRONG! - Compile Error */  
    p = 1;    /* WRONG! - Compile Error */ }
```



Pointers as Return Values

Δείκτες ως Τιμές Επιστροφής

- Οι **Συναρτήσεις** επιτρέπεται να **επιστρέφουν δείκτες**:

```
int *max(int *a, int *b)
{
    if (*a > *b)
        return a;
    else
        return b;
}
```

→ Διαφορετικό από *a

Συμβουλή!

Προτιμάτε ΠΑΝΤΑ να επιστρέψετε τον κωδικό επιτυχίας / αποτυχίας της συνάρτησης (EXIT_SUCCESS, EXIT_ERROR) εκτός από απλές μαθηματικές συναρτήσεις όπως εδώ.

- Μια **κλήση** της συνάρτησης `max` :

```
int *p = NULL, i, j;
...
p = max(&i, &j);
```

Μετά την κλήση, το `p` δείχνει είτε στο `i` ή στο `j`.



Pointers as Return Values

Δείκτες ως Τιμές Επιστροφής

- Μια συνάρτηση θα μπορούσε επίσης να επιστρέφει ένα δείκτη σε μια **εξωτερική μεταβλητή** ή σε μια **εσωτερική static μεταβλητή** (εφόσον αυτές δεν χάνονται με την αποπεράτωση της συνάρτησης)

```
int *f(void) {  
    static int i = 5;  
    return &i;  
}
```

ΟΡΘΟ

- ΠΟΤΕ** μην επιστρέφεται ένα δείκτη σε μια αυτόματη εσωτερική μεταβλητή:

```
int *f(void) {  
    int i = 5;  
    return &i;  
}
```

ΛΑΘΟΣ

Ο χώρος της i δεν υπάρχει μετά την κλήση της f !



Pointers as Return Values

Δείκτες ως Τιμές Επιστροφής

- Οι δείκτες μπορεί να **δείχνουν σε στοιχεία** πινάκων, π.χ., `&a[i]` είναι δείκτης στο στοιχείο i του πίνακα a .
- Μια συνάρτηση θα μπορούσε ακόμη και να επιστρέφει ένα **δείκτη σε ένα από τα στοιχεία ενός πίνακα**.

```
int *find_middle(int a[], int n) {  
    return &a[n/2];  
}
```

- Εάν και όπως αναφέραμε προτιμάμε το return code να το κρατήσουμε για τα μηνύματα λάθους.

