

# Προγραμματισμός I (HY120)

Διάλεξη 4:  
Τελεστές





# Τελεστές: Τελεστής Ανάθεσης

- Το σύμβολο της ανάθεσης είναι το  $=$ 
  - Προσοχή: το σύμβολο ελέγχου ισότητας είναι το  $==$ .
- Η μορφή των προτάσεων ανάθεσης είναι:  
<όνομα> = <έκφραση>
  - 1. Αποτιμάται η έκφραση στο δεξί μέρος.
  - 2. Η τιμή που παράγεται αποθηκεύεται στην μεταβλητή το όνομα της οποίας δίνεται στο αριστερό μέρος.
- Η έκφραση μπορεί να συμπεριλαμβάνει μεταβλητές, στην οποία περίπτωση επιστρέφεται η τιμή που έχει αποθηκευτεί στην μνήμη τους.
- Στο δεξί μέρος μπορεί να εμφανίζεται η ίδια μεταβλητή που εμφανίζεται και το αριστερό.

# Η 2πλή Προσωπικότητα του Τελεστή Ανάθεσης



3

- Η ανάθεση αποτελεί ταυτόχρονα μια έκφραση αποτίμησης που **επιστρέφει την τιμή που ανατίθεται!**
  - Μια έκφραση ανάθεσης μπορεί να χρησιμοποιηθεί ως τμήμα άλλων, πιο πολύπλοκων εκφράσεων.
  - Π.χ. επιτρέπονται εκφράσεις «αλυσιδωτής» ανάθεσης τιμών (από δεξιά προς τα αριστερά), όπου όλες οι μεταβλητές παίρνουν την τιμή που εμφανίζεται στο δεξί μέρος.



```
int i,j,k;

i = 1;          /* i γίνεται 1 */

j = 1+1;       /* j γίνεται 2 */

k = i+j;       /* k γίνεται 3 */

i = k = j;     /* i,k γίνονται 2 */

j = (i=3) + k; /* i γίνεται 3, j γίνεται 5 */

i = i+1;       /* i γίνεται 4 */
```

# Τελεστής Ανάθεσης: Λίγοι Ιδιωματισμοί



5

- Πολλές φορές χρησιμοποιούμε την ανάθεση για να αλλάξουμε την τιμή μιας μεταβλητής σε σχέση με την παλιά τιμή της (ίδιας μεταβλητής).
  - Για το πετύχουμε αυτό, γράφουμε:  
 $\langle \text{όνομα} \rangle = \langle \text{όνομα} \rangle \langle \text{op} \rangle \langle \text{έκφραση} \rangle$   
όπου  $\langle \text{op} \rangle$  ένας τελεστής.
  - Το ίδιο αποτέλεσμα μπορεί να επιτευχθεί με χρήση της «σχετικής» ανάθεσης  $\langle \text{op} \rangle =$ , ως εξής:  
 $\langle \text{όνομα} \rangle \langle \text{op} \rangle = \langle \text{έκφραση} \rangle$ 
    - Προσοχή στις προτεραιότητες...
    - ... η έκφραση που εμφανίζεται στα δεξιά αποτιμάται πριν εφαρμοστεί ο τελεστής  $\langle \text{op} \rangle$

# Δηλαδή;



```
int i,j,k;

i = 1;

j = 2;

i += 2;          /* i γίνεται 3 */

j *= i+1;        /* j γίνεται 8 */

k = (i+=j) + 1;  /* i γίνεται 11, k γίνεται 12
*/
```

# Κι άλλη Ειδική Περίπτωση: Τελεστές Αυξομείωσης



- Μια ειδική περίπτωση ανάθεσης είναι η αύξηση ή μείωση της τιμής μιας μεταβλητής κατά 1:

$\langle \text{όνομα} \rangle = \langle \text{όνομα} \rangle \langle \text{op} \rangle 1$

$\langle \text{όνομα} \rangle \langle \text{op} \rangle = 1$

όπου  $\langle \text{op} \rangle$  ο τελεστής + ή -

- Παρόμοιο αποτέλεσμα μπορεί να επιτευχθεί με χρήση των τελεστών ++ ή -- :

$\langle \text{όνομα} \rangle ++$  ή  $\langle \text{όνομα} \rangle --$

$++\langle \text{όνομα} \rangle$  ή  $--\langle \text{όνομα} \rangle$

- Όταν ο τελεστής εμφανίζεται πριν το όνομα της μεταβλητής, τότε ως αποτέλεσμα της έκφρασης επιστρέφεται η νέα τιμή της μεταβλητής.
- Όταν εμφανίζεται μετά, ως αποτέλεσμα της έκφρασης επιστρέφεται η παλιά τιμή της μεταβλητής



```
int i,j,k;

i = 0;

i++;          /* i γίνεται 1 */

j = i++;     /* j γίνεται 1, i γίνεται 2 */

k = --j;     /* k γίνεται 0, j γίνεται 0 */

i = (k++) + 1; /* i γίνεται 1, k γίνεται 1 */

i = (++k) + 1; /* i γίνεται 3, k γίνεται 2 */
```



# Προσοχή, προσοχή, προσοχή!!!



- Καμιά φορά τα φαινόμενα απατούν
  - Φροντίστε να ξέρετε τι ακριβώς κάνετε...

9

```
int i;
i = 0;
i = (i++);

i = 0;
i = (++i);

i = 0;
i = (i=i+1);

i = 0;
i = (i++) + (i++);

i = 0;
i = (++i) + (++i);

i = 0;
i = (i=i+1) + (i=i+1);
```

```
gcc 4.1.2 (SUSE)
/* i γίνεται 1 */
/* i γίνεται 1 */
/* i γίνεται 1 */
/* i γίνεται 2 */
/* i γίνεται 4 */
/* i γίνεται 4 */
```

```
gcc 3.4.2 (mingw32)
/* i γίνεται 0 */
/* i γίνεται 1 */
/* i γίνεται 1 */
/* i γίνεται 2 */
/* i γίνεται 4 */
/* i γίνεται 3 */
```

# Αριθμητικοί Τελεστές για `int` / `float` / `double`



10

- `+` πρόσθεση δύο τιμών
- `-` αφαίρεση δύο τιμών
- `*` πολλαπλασιασμός δύο τιμών
- `/` διαίρεση δύο τιμών
- `%` υπόλοιπο διαίρεσης δύο τιμών (μόνο για ακέραιους)
  - $x = y * (x / y) + x \% y$ , όμως δεν ισχύει πάντα η «μαθηματική» ιδιότητα του υπολοίπου ( $\geq 0$ ), π.χ. όταν ο αριθμητής είναι αρνητικός.
  - Π.χ. `-5%4` ή `-5%-4`.
- Η αποτίμηση γίνεται από αριστερά προς τα δεξιά.
- Τα `/` και `%` έχουν μεγαλύτερη προτεραιότητα αποτίμησης σε σχέση με τα `+` και `-`.

# Πράξεις & Τελεστές σε Επίπεδο Bits



11

- $\&$  δυαδικό «and»
- $|$  δυαδικό «or»
- $\wedge$  δυαδικό «xor»
- $\sim$  δυαδικό «not»
- $\ll$  αριστερή ολίσθηση (LSB (least significant bit)  $\rightarrow$  MSB (most significant bit))
  - Στην αριστερή ολίσθηση τα «λιγότερο σημαντικά» bits παίρνουν πάντα την τιμή 0
- $\gg$  δεξιά ολίσθηση (MSB (most significant bit)  $\rightarrow$  LSB (least significant bit))
  - Στη δεξιά ολίσθηση τα «περισσότερο σημαντικά» bits παίρνουν την τιμή
    - Του περισσότερο σημαντικού bit (arithmetic shift) αν το όρισμα ερμηνεύεται ως signed
    - 0 (logical shift), αν το όρισμα ερμηνεύεται ως unsigned.



```
char a,b,c;
unsigned char d;

a = 0x61;      /* a γίνεται 01100001 */
b = 0x62;      /* b γίνεται 01100010 */
c = a|b;       /* c γίνεται 01100011 */
c = a&b;       /* c γίνεται 01100000 */
c = a^b;       /* c γίνεται 00000011 */
d = c = ~a;    /* d,c γίνεται 10011110 */
d = d>>3;     /* d γίνεται 00010011 */
c = c>>3;     /* c γίνεται 11110011 */
```

# Γρήγορος Πολλαπλασιασμός / Διαίρεση



13

- Με τον τελεστή ολίσθησης bits μπορούμε να υλοποιήσουμε γρήγορες πράξεις πολλαπλασιασμού και διαίρεσης με τιμές που είναι δυνάμεις του 2:

$$v = v \ll i; \quad /* \quad v = v * 2^i \quad */$$

$$v = v \gg i; \quad /* \quad v = v / 2^i \quad */$$

- Και φυσικά μπορούμε να υπολογίσουμε εύκολα τις δυνάμεις του 2:

$$v = 1 \ll i; \quad /* \quad v = 2^i \quad */$$



```
short int i;

i = 5;          /* i γίνεται 00000000 00000101 */
i = i<<4;       /* i γίνεται 00000000 01010000 */
i = i>>2;       /* i γίνεται 00000000 00010100 */
i = 1<<3;       /* i γίνεται 00000000 00001000 */
i = 1<<8;       /* i γίνεται 00000001 00000000 */
```



# Σχεσιακοί και Λογικοί Τελεστές

- `==`, `!=` ισότητα, ανισότητα
  - `>`, `>=` μεγαλύτερο, μεγαλύτερο ίσο
  - `<`, `<=` μικρότερο, μικρότερο ίσο
  - `!` λογική άρνηση
- 
- Οι σχεσιακοί και λογικοί τελεστές χρησιμοποιούνται για την κατασκευή λογικών εκφράσεων (συνθηκών).
  - Δεν υπάρχει λογικός τύπος (boolean).
  - Το αποτέλεσμα μιας λογικής έκφρασης είναι 0 ή 1 (για ψευδές ή αληθές), και μπορεί να χρησιμοποιηθεί και ως ακέραιος
    - Κλασική πηγή λαθών στην C.
    - Η τιμή 0 ερμηνεύεται ως «ψευδές» (false)
    - Οποιαδήποτε τιμή διάφορη του 0 ως «αληθές» (true).



```
int a=1, b=2, c;

c = (a == b);           /* c γίνεται 0 */

c = (a != b);          /* c γίνεται 1 */

c = (a <= b);          /* c γίνεται 1 */

c = ((c + a) != b);    /* c γίνεται 0 */

c = (!a == !b);        /* c γίνεται 1 */

c = (a != b) + !(a == b) /* c γίνεται 2 */

c = !( (a != b) + !(a == b) ); /* c γίνεται 0 */
```





# Λογικοί Τελεστές

- `||` λογικό «ή»
- `&&` λογικό «και»
- Οι λογικοί σύνδεσμοι χρησιμοποιούνται για την κατασκευή σύνθετων λογικών εκφράσεων.
- Η αποτίμηση των λογικών εκφράσεων γίνεται από «τα αριστερά προς τα δεξιά» και **μόνο όσο χρειάζεται για να διαπιστωθεί το τελικό αποτέλεσμα της έκφρασης (conditional evaluation)**.
  - Για το `||` η αποτίμηση σταματά μόλις η ενδιάμεση τιμή γίνει διάφορη του 0
  - Για το `&&` η αποτίμηση σταματά μόλις η ενδιάμεση τιμή γίνει 0.
  - Προσοχή στις παρενέργειες!



```
int a=1, b=0, c;

c = a&&b;          /* c γίνεται 0 */

c = (a==1) || (b==1); /* c γίνεται 1 */

c = (a++>1);      /* c γίνεται 0,
                  a γίνεται 2 */

c = b && (a++);    /* c γίνεται 0,
                  a παραμένει 2 */

c = (b++) && (a++); /* c γίνεται 0,
                  b γίνεται 1,
                  a παραμένει 2 */

c = (--b) || (a++); /* c γίνεται 1,
                  b γίνεται 0,
                  a γίνεται 3 */
```



# Παρενέργειες

- **Παρενέργεια**: αλλαγή τιμής μιας μεταβλητής χωρίς αυτό να είναι εύκολα ορατό από τον κώδικα.
- Η τεχνική αποτίμησης λογικών εκφράσεων μπορεί να οδηγήσει σε κώδικα με παρενέργειες,
  - π.χ. `<lexpr> && (a++)`
    - οδηγεί σε αλλαγή της τιμής της μεταβλητής `a` μόνο όταν η `<lexpr>` αποτιμάται ως αληθής
    - Δεν είναι εύκολο να εκτιμηθεί διαβάζοντας τον (υπόλοιπο) κώδικα.
- Ο προγραμματισμός με παρενέργειες θεωρείται κακό στυλ
  - Οδηγεί σε κώδικα που είναι αρκετά δύσκολο να κατανοηθεί.



# Λοιποί Τελεστές

- **<lexpr>?<expr1>:<expr2>** αποτιμά την έκφραση `lexpr` και εφόσον η τιμή της είναι διάφορη του 0 αποτιμά και επιστρέφει το αποτέλεσμα της έκφρασης `expr1`, διαφορετικά αποτιμά και επιστρέφει το αποτέλεσμα της έκφρασης `expr2`.
- **`expr1, expr2, ..., exprn`** αποτιμά τις εκφράσεις `expr1`, `expr2` μέχρι και `exprn`, «από αριστερά προς τα δεξιά», και επιστρέφει το αποτέλεσμα της τελευταίας.

● Π.χ.:

```
int a = 1, b = 2, c, d;
```

```
c = (a<b) ? a : b;
```

```
d = (c=a, a=b, b=c, c=0);
```

```
/* c == 1 */
```

```
/* a,b,c,d == 2,1,0,0 */
```



# Προτεραιότητα Τελεστών

μέγιστη	( ) [ ] . ->
	! ~ ++ -- + - * & (cast) (ατομικοί)
	* / %
	+ -
	<< >>
	> >= < <=
	==, !=
	&
	>
	&&
	? :
	= += -= *= /= %= &=  = <<= >>=
	,
Σειρά αποτίμησης	
ελάχιστη	