

Προγραμματισμός Ι

Τελεστές

Πανεπιστήμιο Πελοποννήσου
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Νικόλαος Δ. Τσελίκας

Ο τελεστής εκχώρησης =

- Ο τελεστής = χρησιμοποιείται για την απόδοση τιμής (ή αλλιώς ανάθεση τιμής) σε μία μεταβλητή
- Π.χ. με την εντολή:

`a = 10;`

η τιμή της μεταβλητής `a` γίνεται 10 (μπορούμε επίσης να πούμε ότι στη μεταβλητή `a` εκχωρήθηκε η τιμή 10)

ενώ με την εντολή:

`a = k;`

η τιμή της μεταβλητής `a` γίνεται ίση με την τιμή της μεταβλητής `k` (μπορούμε επίσης να πούμε ότι στη μεταβλητή `a` εκχωρήθηκε η τιμή της μεταβλητής `k`)

- Αν ο τελεστής = χρησιμοποιείται πολλές φορές σε μία εντολή εκχώρησης, τότε η τελική τιμή εκχώρησης αποθηκεύεται σε όλες τις μεταβλητές (ο τελεστής εφαρμόζεται από δεξιά προς τα αριστερά)
- Π.χ. με την εντολή:

`int a,b,c;`

`a = b = c = 10;`

οι τιμές των μεταβλητών `a`, `b` και `c` γίνονται ίσες με 10

Παρατηρήσεις (I)



ΠΡΟΣΟΧΗ!!!

Τονίζουμε ότι ο τελεστής εκχώρησης = εφαρμόζεται από δεξιά προς τα αριστερά

- Συνεπώς, τί θα εμφανίσει το παρακάτω πρόγραμμα ???

```
int a, b, c;  
a = b = 10 = c;  
printf("%d %d %d\n", a, b, c);
```

- Το πρόγραμμα δεν θα "τρέξει", και ο compiler θα εντοπίσει συντακτικό λάθος (δεδομένου ότι γίνεται προσπάθεια εκχώρησης της τιμής της μεταβλητής c στην "σταθερή τιμή" 10... (πράγμα αδύνατον)
- Μην συγχέετε λοιπόν την ισότητα από τα "μαθηματικά", με τον τελεστή ανάθεσης...

Παρατηρήσεις (II)



Ο δεξιός τελεστέος (right operand) πρέπει να είναι τύπου `rvalue`, δηλ. μία έκφραση που έχει τιμή, όπως μία σταθερά ή μία μεταβλητή



Ο αριστερός τελεστέος (left operand) πρέπει να είναι τύπου `lvalue`

- ♦ Ο τύπος `lvalue` αναφέρεται σε κάτι αποθηκευμένο στη μνήμη που μπορεί να μεταβάλλεται, όπως μία μεταβλητή
- ♦ Ο αριστερός τελεστέος απαγορεύεται να είναι σταθερά ή το αποτέλεσμα ενός υπολογισμού, π.χ.:

```
10 = a; /* Δεν επιτρέπεται. */  
a+b = 20; /* Δεν επιτρέπεται. */
```

- Στις παραπάνω περιπτώσεις ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους της μορφής: `"error '=' : left operand must be lvalue"`

Παρατηρήσεις (III)

- Αν ο τύπος της μεταβλητής δεν είναι ίδιος με τον τύπο της τιμής, τότε, εφόσον η μετατροπή είναι δυνατή, η τιμή πρώτα μετατρέπεται στον τύπο της μεταβλητής και μετά εκχωρείται σε αυτήν

Π.χ.

```
int a;  
float b;  
b = a = 10.96;
```

- Ποιες τιμές αποθηκεύονται στις μεταβλητές a και b???

Παρατηρήσεις (IV)

- Σε περίπτωση **υπερχείλισης (overflow)**, σε υπολογισμούς με τιμές προσημασμένων μεταβλητών όπως π.χ. κατά την πρόσθεση δύο μεγάλων ακεραίων, το αποτέλεσμα δεν θα έχει νόημα και δεν είναι δυνατόν να προσδιοριστεί
- Για την **αποφυγή υπερχειλίσης** μπορούμε να ελέγχουμε την πιθανότητα να γίνει υπερχειλίση, πριν αυτή γίνει, π.χ. θεωρήστε ότι οι `a`, `b` και `c` είναι όλες `int` μεταβλητές και η `b` είναι θετική
- Θα μπορούσαμε πριν αναθέσουμε την τιμή `a+b` στη `c`, να γράψουμε:


```
if(a > INT_MAX - b) /* Η INT_MAX ορίζεται στο <limits.h>. */
{
    printf("Error: overflow case\n");
    return;
}
c = a+b;
```



Όταν πραγματοποιείτε αριθμητικές πράξεις, να είστε σίγουροι ότι δεν θα προκύψει υπερχειλίση για να αποφύγετε δυσάρεστες καταστάσεις

Αριθμητικοί τελεστές (1/2)


- Οι μαθηματικοί τελεστές $+$, $-$, $*$, $/$ χρησιμοποιούνται για την εκτέλεση των γνωστών μαθηματικών πράξεων

 Αν και οι δύο τελεστέοι είναι ακέραιοι, τότε ο τελεστής $/$ ισοδυναμεί με το πηλίκο της ακέραιας διαίρεσης (αποκόπτεται το δεκαδικό μέρος)

- Π.χ.,

```
int a = 3, b = 2, c;  
c = a/b;
```

η τιμή της έκφρασης a/b που ανατίθεται στη μεταβλητή c είναι 1 και όχι 1.5

 Αν, έστω και ένας απ' τους τελεστέους είναι μεταβλητή ή σταθερά δεκαδικού τύπου (`float` ή `double`), εκτελείται κανονικά η διαίρεση και το δεκαδικό μέρος δεν αποκόπτεται, π.χ.:

```
int a = 8;  
float b = 5;
```

το αποτέλεσμα a/b είναι 1.6, αφού η μεταβλητή b είναι `float` και το αποτέλεσμα $a/10.0$ είναι 0.8, αφού και η σταθερά (10.0) έχει δεκαδικό μέρος επίσης

Αριθμητικοί τελεστές (2/2)

- Ο τελεστής `%` χρησιμοποιείται για τον υπολογισμό **του υπολοίπου της ακεραίας διαίρεσης** δύο ακεραίων
- Π.χ. στο επόμενο παράδειγμα :

```
int a, b, c, d;  
a = 11;  
b = 3;  
c = a%b;  
d = a%c%b;
```

η τιμή της μεταβλητής `c` είναι 2 και η τιμή της μεταβλητής `d` είναι 1



ΠΡΟΣΟΧΗ!!! Ο τελεστής `%` μπορεί να εφαρμοστεί μόνο μεταξύ ακεραίων αριθμών, αλλιώς ο μεταγλωττιστής εμφανίζει μήνυμα λάθους

Παρατηρήσεις (I)

- Αν ο δεξιός τελεστέος κατά τη χρήση του τελεστή / ή % έχει τιμή 0, το αποτέλεσμα είναι ακαθόριστο (undefined), π.χ. το πρόγραμμα μπορεί ακόμα και να τερματίσει
- Αν οι τελεστέοι είναι αρνητικοί, το αποτέλεσμα εξαρτάται απ' την υλοποίηση του μεταγλωττιστή
- Π.χ., το αποτέλεσμα του: $-8/6$ μπορεί να είναι είτε -1 είτε -2 , κι εξαρτάται απ' την υλοποίηση αν στρογγυλοποιεί προς τα πάνω ή προς τα κάτω το αποτέλεσμα
- Ομοίως, το αποτέλεσμα του: $-8\%6$ μπορεί να είναι είτε 2 είτε -2 , κι εξαρτάται επίσης απ' την υλοποίηση
- Γι' αυτόν τον λόγο, αν θέλετε να ελέγξετε αν το n είναι περιττός, μη γράφετε κάτι σαν αυτό:

```
if ((n % 2) == 1)
```

- Αν το n είναι περιττός, αλλά αρνητικός, το αποτέλεσμα σε κάποιες υλοποιήσεις μεταγλωττιστών μπορεί να είναι -1 , οπότε ένας τέτοιος έλεγχος αποτυγχάνει, δηλαδή είναι πιο ασφαλές να γράψετε:

```
if ((n % 2) != 0)
```

Προγραμματισμός I

Παρατηρήσεις (II)

- Οι πέντε αυτοί τελεστές λέγονται και «**δυναδικοί**» (**binary**), επειδή απαιτούν δύο τελεστέους
- Η C παρέχει και τους **μοναδιαίους** (**unary**) τελεστές + και - που απαιτούν μόνο έναν τελεστέο
 - ◆ Ο μοναδιαίος τελεστής + δεν επιφέρει κάποιο αποτέλεσμα στον τελεστέο
 - ◆ Το αποτέλεσμα της χρήσης του μοναδιαίου τελεστή - είναι ο αντίθετος του τελεστέου
- Όταν σε κάποια έκφραση εμπλέκονται και μοναδιαίοι και δυναδικοί τελεστές + και -, καθορίζει ο μεταγλωττιστής το πώς εφαρμόζονται, π.χ.:

```
int a = +20; /* Ο μοναδιαίος τελεστής + δεν έχει κάποια επίδραση. */  
a = -10; /* Το - είναι εδώ ο μοναδιαίος τελεστής. */  
int b = -(a-5); /* Το «μέσα -» αντιστοιχεί στον δυναδικό τελεστή ενώ  
το «έξω -» στον μοναδιαίο, αντίστοιχα. */  
-a; /* Δεν επιφέρει κάποιο αποτέλεσμα στη μεταβλητή a (δεν γίνεται  
κάποια ανάθεση), άρα η τιμή της a παραμένει ως έχει. */
```

- Δηλ, η μεταβλητή b αποκτά την τιμή 15 και η a την τιμή -10

Ο τελεστής αύξησης ++

- Ο τελεστής αύξησης ++ μπαίνει πριν ή μετά από το όνομα μίας μεταβλητής
- Σε κάθε περίπτωση η τιμή της μεταβλητής αυξάνεται κατά ένα

```
#include <stdio.h>
int main(void)
{
    int a = 4;
    a++; /* Ισοδύναμο με a = a+1; */
    printf("Num = %d\n",a);
    return 0;
}
```

Ο τελεστής αύξησης ++ στην εκχώρηση

- Όταν χρησιμοποιούμε τον τελεστή αύξησης ++ σε κάποια εντολή εκχώρησης, τότε:
 - ♦ Αν ο τελεστής χρησιμοποιείται μετά το όνομα της μεταβλητής, τότε πρώτα χρησιμοποιείται η τρέχουσα τιμή της μεταβλητής και μετά αυτή αυξάνεται κατά ένα
 - ♦ Αν ο τελεστής χρησιμοποιείται πριν το όνομα της μεταβλητής, τότε πρώτα αυξάνεται η τιμή της μεταβλητής κατά ένα και μετά αυτή χρησιμοποιείται

```
#include <stdio.h>
int main(void)
{
    int a,b;

    a = 4;
    b = a++;
    printf("a = %d b = %d\n",a,b);
    return 0;
}
```

Έξοδος: a = 5 b = 4

```
#include <stdio.h>
int main(void)
{
    int a,b;

    a = 4;
    b = ++a;
    printf("a = %d b = %d\n",a,b);
    return 0;
}
```

Έξοδος: a = 5 b = 5

Ο τελεστής μείωσης --

- Ο τελεστής μείωσης -- μπαίνει πριν ή μετά από το όνομα μίας μεταβλητής
- Σε κάθε περίπτωση η τιμή της μεταβλητής μειώνεται κατά ένα

```
#include <stdio.h>
int main(void)
{
    int a = 4;
    a--; /* Ισοδύναμο με a = a-1; */
    printf("Num = %d\n",a);
    return 0;
}
```

- Για τον τελεστή μείωσης ισχύουν ακριβώς οι ίδιοι κανόνες που παρουσιάστηκαν για τον τελεστή αύξησης

Παραδείγματα

- Ποια είναι η έξοδος του καθενός προγράμματος?

```
#include <stdio.h>
int main(void)
{
    double a = 3.45;

    printf("%.2f\n", a++);
    printf("%.2f\n", ++a);
    return 0;
}
```

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 2, c = 3;
    printf("%d\n", (++a)-(b--)+(--c));
    return 0;
}
```

Το πρόγραμμα εμφανίζει αρχικά 3.45, και μετά η τιμή της μεταβήτης *a* αυξάνεται.

Στη συνέχεια, η τιμή της *a* πρώτα αυξάνεται, συνεπώς, το πρόγραμμα εμφανίζει 5.45

Με το *++a* πρώτα αυξάνει την τρέχουσα τιμή της *a*, και στη συνέχεια η νέα αυτή τιμή χρησιμοποιείται. Με το *b--* χρησιμοποιείται πρώτα η τρέχουσα τιμή της *b* και μετά μειώνεται. Τέλος, με το *--c* η τιμή της *c* πρώτα μειώνεται και η νέα τιμή χρησιμοποιείται στην έκφραση. Συνεπώς, το αποτέλεσμα της έκφρασης είναι: $2-2+2 = 2$ ενώ οι τιμές των *a*, *b*, και *c* γίνονται 2, 1, and 2, αντίστοιχα.

Παρατηρήσεις



Μην εφαρμόζετε τους τελεστές ++ και -- ταυτόχρονα στην ίδια μεταβλητή και μην δημιουργείτε εκφράσεις των οποίων η τελική τιμή εξαρτάται από τη σειρά που μπορεί να εφαρμοστούν οι επί μέρους τελεστές

- Π.χ., το αποτέλεσμα της έκφρασης:

```
a * a++;
```

εξαρτάται απ' την υλοποίηση του μεταγλωττιστή, δηλαδή ο ίδιος ο μεταγλωττιστής θα καθορίσει αν η τιμή της μεταβλητής a πρέπει πρώτα να αυξηθεί και μετά να γίνει ο πολλαπλασιασμός ή το αντίστροφο

Τελεστές Σύγκρισης (I)

- Οι τελεστές σύγκρισης `>`, `>=`, `<`, `<=`, `!=`, `==`, χρησιμοποιούνται για τη σύγκριση των τιμών που έχουν δύο εκφράσεις
- Συνήθως χρησιμοποιούνται σε εντολές ελέγχου (π.χ. στην εντολή `if`) και σε επαναληπτικούς βρόχους (π.χ. στις εντολές `for`, `while` ή `do-while`)

■ Π.χ.

- `if (a > 10)` ο τελεστής `>` χρησιμοποιείται για να ελέγξουμε αν η τιμή της μεταβλητής `a` είναι μεγαλύτερη από το 10.
- `if (a >= 10)` ο τελεστής `>=` χρησιμοποιείται για να ελέγξουμε αν η τιμή της μεταβλητής `a` είναι μεγαλύτερη ή ίση από το 10.
- `if (a < 10)` ο τελεστής `<` χρησιμοποιείται για να ελέγξουμε αν η τιμή της μεταβλητής `a` είναι μικρότερη από το 10.
- `if (a <= 10)` ο τελεστής `<=` χρησιμοποιείται για να ελέγξουμε αν η τιμή της μεταβλητής `a` είναι μικρότερη ή ίση από το 10.
- `if (a != 10)` ο τελεστής `!=` χρησιμοποιείται για να ελέγξουμε αν η τιμή της μεταβλητής `a` είναι διαφορετική από το 10.
- `if (a == 10)` ο τελεστής `==` χρησιμοποιείται για να ελέγξουμε αν η τιμή της μεταβλητής `a` είναι ίση με το 10.

Τελεστές Σύγκρισης (II)

Tip Μία **έκφραση** χαρακτηρίζεται **αληθής (true)**, όταν η τιμή της είναι διαφορετική από το μηδέν, ενώ - αν είναι μηδέν- χαρακτηρίζεται **ψευδής (false)**

- Το αποτέλεσμα της έκφρασης στην οποία χρησιμοποιείται κάποιος τελεστής σύγκρισης είναι **1** (όταν η έκφραση είναι **αληθής - true**) ενώ το αποτέλεσμα είναι **0** (όταν η έκφραση είναι **ψευδής - false**)

- Π.χ. το αποτέλεσμα της έκφρασης $(a > 10)$ είναι ένα (**1**) μόνο αν η τιμή της μεταβλητής **a** είναι μεγαλύτερη από το **10**, αλλιώς είναι μηδέν (**0**)

- Ποια είναι η έξοδος του διπλανού προγράμματος:

Έξοδος: 1 0

```
#include <stdio.h>
int main(void)
{
    int a = 3, b = 5;

    a = (a > 3) + (b <= 5);
    a = (a == 3) + ((b-2) >= 3);
    b = (b != 5);
    printf("%d %d\n", a, b);
    return 0;
}
```

Παρατηρήσεις



Μην συγχέετε τον τελεστή **ελέγχου ισότητας** (**==**) με τον τελεστή **εκχώρησης** (**=**)

- Ο τελεστής **==** χρησιμοποιείται για να ελέγξουμε αν δύο εκφράσεις έχουν την ίδια τιμή, ενώ ο τελεστής **=** χρησιμοποιείται για να αναθέσουμε μία τιμή σε μία μεταβλητή
- Π.χ., η τιμή της έκφρασης:

$$a == 10$$

είναι **1**, μόνο η τιμή της μεταβλητής **a** είναι **10**, αλλιώς είναι **0**

ενώ η τιμή της έκφρασης:

$$a = 10$$

είναι πάντοτε **1** και η έκφραση εκχωρεί την τιμή **10** στη μεταβλητή **a** (η εκχώρηση μη μηδενικής τιμής είναι πάντα αληθής έκφραση, ενώ η εκχώρηση του μηδενός είναι πάντα ψευδής έκφραση, αντίστοιχα)

Ο τελεστής !

- Ο τελεστής ! είναι μοναδιαίος, δηλαδή εφαρμόζεται σε έναν μόνο τελεστέο
- Αν μία έκφραση `exp` είναι **αληθής** (δηλαδή έχει **μη μηδενική τιμή**), τότε το αποτέλεσμα της πράξης `!exp` είναι μηδέν (0)
- Αν μία έκφραση `exp` είναι **ψευδής** (δηλαδή έχει **μηδενική τιμή**), τότε το αποτέλεσμα της πράξης `!exp` είναι ένα (1)

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος:

```
#include <stdio.h>
int main(void)
{
    int a = 4;
    printf("Num = %d\n", !a);
    return 0;
}
```

Έξοδος: Num = 0

- Ποια είναι η έξοδος του παρακάτω προγράμματος:

```
#include <stdio.h>
int main(void)
{
    int a = 4;
    printf("Num = %d\n", !!a);
    return 0;
}
```

Έξοδος: Num = 1

Παρατηρήσεις

- Συνήθως, ο τελεστής `!` χρησιμοποιείται σε συνθήκες ελέγχου στην εντολή `if`

- Π.χ.
η εντολή:

```
if (!a)
```

είναι ισοδύναμη με

```
if (a == 0)
```

και η εντολή:

```
if (a)
```

είναι ισοδύναμη με

```
if (a != 0)
```

Συνδυαστικοί τελεστές

- Οι συνδυαστικοί τελεστές χρησιμοποιούνται για να γραφούν μαθηματικές εκφράσεις με πιο σύντομο τρόπο, βάσει του παρακάτω τύπου:

```
expr1 op= expr2;
```

όπου συνήθως ο τελεστής **op** είναι κάποιος από τους αριθμητικούς τελεστές $+$, $-$, $*$, $\%$, $/$ ή κάποιος από τους τελεστές bit που θα δούμε παρακάτω ($\&$, \wedge , $|$, \ll , \gg). Η παραπάνω έκφραση είναι ισοδύναμη με:

```
expr1 = expr1 op (expr2);
```

- Π.χ. η έκφραση:

```
a += b;
```

είναι ισοδύναμη με:

```
a = a + b;
```

ενώ η έκφραση:

```
a *= 3;
```

είναι ισοδύναμη με:

```
a = a * 3;
```

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος:

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 2;

    a += 6;
    a *= b+3;
    a -= b+8;
    a /= b;
    a %= b+1;
    printf("Num = %d\n", a);
    return 0;
}
```

Έξοδος: Num = 2

Λογικοί τελεστές

◆ Ο τελεστής &&

- Η τιμή μίας έκφρασης που περιέχει τον τελεστή && είναι ένα (1), δηλαδή **αληθής**, μόνο αν όλοι οι όροι της έκφρασης είναι αληθείς
- η τιμή της έκφρασης που περιέχει τον τελεστή && είναι μηδέν (0), δηλαδή **ψευδής**, αν έστω και ένας όρος έχει ψευδή τιμή
- Ο τελεστής && εφαρμόζει δηλαδή τη λογική πράξη ΚΑΙ (λογική πράξη AND) μεταξύ των όρων στους οποίους εφαρμόζεται

◆ Ο τελεστής ||

- Μία έκφραση που περιέχει τον τελεστή || είναι ένα (1), δηλαδή **αληθής**, αν έστω και ένας όρος της έκφρασης είναι αληθής
- Μία έκφραση που περιέχει τον τελεστή || είναι μηδέν (0), δηλαδή **ψευδής**, αν κανένας όρος της έκφρασης δεν είναι αληθής
- Ο τελεστής || εφαρμόζει δηλαδή τη λογική πράξη Ή (λογική πράξη OR) μεταξύ των όρων στους οποίους εφαρμόζεται

Παραδείγματα

- Η έκφραση $(10 == 10) \ \&\& \ (5 > 3)$ είναι **αληθής**, γιατί και οι δύο όροι της έκφρασης είναι αληθείς
- Αν γράψαμε:
 $a = (10 == 10) \ \&\& \ (5 > 3);$ τότε η τιμή της μεταβλητής a θα γινόταν ίση με 1

- Η έκφραση $(10 == 10) \ \&\& \ (5 > 3) \ \&\& \ (13 < 8)$ είναι **ψευδής**, γιατί υπάρχει ένας όρος που έχει ψευδή τιμή
- Αν γράψαμε:
 $a = (10 == 10) \ \&\& \ (5 > 3) \ \&\& \ (13 < 8);$ τότε η τιμή της μεταβλητής a θα γινόταν ίση με 0

Παραδείγματα

- Η έκφραση $(10 == 10) \ || \ (3 > 5)$ είναι **αληθής**, γιατί ένας όρος της έκφρασης είναι αληθής

- Αν γράψαμε:

$a = (10 == 10) \ || \ (3 > 5);$ τότε η τιμή της μεταβλητής a θα γινόταν ίση με 1

- Η έκφραση $(10 != 10) \ || \ (3 > 5)$ είναι **ψευδής**, γιατί δεν υπάρχει κάποιος όρος που να είναι αληθής

- Αν γράψαμε:

$a = (10 != 10) \ || \ (3 > 5);$ τότε η τιμή της μεταβλητής a θα γινόταν ίση με 0

Παρατηρήσεις

- Αν ο όρος που ελέγχεται σε μία έκφραση με τον τελεστή $\&\&$ έχει ψευδή τιμή, τότε ο μεταγλωττιστής δεν ελέγχει τους υπόλοιπους όρους και θέτει κατευθείαν την τιμή της συνολικής έκφρασης ίση με 0
- Αντίστοιχα, αν ο όρος που ελέγχεται σε μία έκφραση με τον τελεστή $||$ είναι αληθής, τότε ο μεταγλωττιστής δεν ελέγχει τους υπόλοιπους όρους και θέτει κατευθείαν την τιμή της συνολικής έκφρασης ίση με 1

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος?

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 4, c = 4;

    printf("%d\n", (a-1) || (b-- == c));
    printf("%d", !(a > b) && (a-1 == b-3) && (c%2 || (a+b-c)));
    return 0;
}
```

Έξοδος: 1
0

Ο τελεστής ,

- Ο τελεστής κόμμα (,) διαχωρίζει επί μέρους εκφράσεις οι οποίες αποτελούν μία έκφραση:

`expr1, expr2, expr3, ...`

- Επειδή ο τελεστής κόμμα (,) παρουσιάζει αριστερή συσχέτιση (είναι left associative) οι επί μέρους εκφράσεις εκτελούνται διαδοχικά από αριστερά προς τα δεξιά
- Δηλαδή, πρώτη εκτελείται η `expr1`, μετά η `expr2`, μετά η `expr3`, έως και την τελευταία έκφραση
- Ο τύπος και η τιμή ολόκληρης της έκφρασης είναι οι αντίστοιχοι της τελευταίας εκτελεσθείσας έκφρασης

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος?

```
#include <stdio.h>
int main(void)
{
    int b;
    b = 20, b = b+30, printf("%d\n", b);
    return 0;
}
```

Έξοδος: 50

Παρατηρήσεις



Δεδομένου ότι η χρήση του τελεστή κόμμα (,) μπορεί να οδηγήσει σε δυσνόητο και περίπλοκο κώδικα, καλό θα ήταν να αποφεύγεται

- Ο τελεστής κόμμα (,) χρησιμοποιείται κυρίως στην εντολή `for`, π.χ.:

```
int a, b;
```

```
for (a = 1, b = 2; b < 10; a++, b++)
```

- Στο παραπάνω κομμάτι κώδικα στην πρώτη έκφραση εντός της παρένθεσης, πρώτα η τιμή της `a` γίνεται 1 και μετά η τιμή της `b` γίνεται 2, ενώ, στην τρίτη έκφραση εντός της παρένθεσης πρώτα εκτελείται η εντολή `a++` και μετά η `b++`
- Σημειώστε ότι τα κόμματα που χρησιμοποιούνται για να χωρίσουν τις παραμέτρους μίας συνάρτησης ή τις μεταβλητές κατά τη δήλωση μεταβλητών του ίδιου τύπου δεν είναι «τελεστές κόμμα», αλλά απλοί διαχωριστές (`separators`)

Ο τελεστής sizeof

- Ο τελεστής `sizeof` υπολογίζει τις οκτάδες που δεσμεύει στη μνήμη του υπολογιστή ο τύπος δεδομένων ή η μεταβλητή που δηλώνεται στις παρενθέσεις του
- Π.χ.

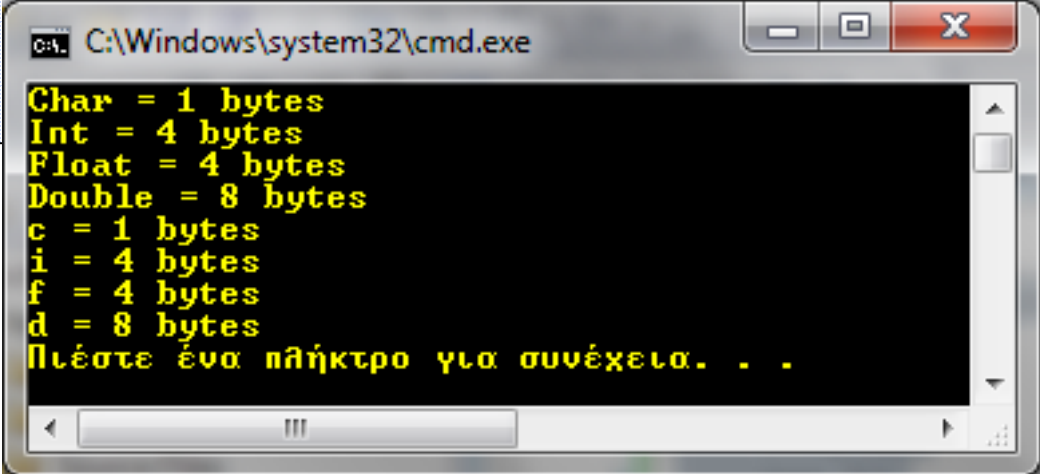
```
#include <stdio.h>
int main(void)
{
    char c;
    int i;
    float f;
    double d;
```

```
    printf("Char = %d bytes\n", sizeof(char));
    printf("Int = %d bytes\n", sizeof(int));
    printf("Float = %d bytes\n", sizeof(float));
    printf("Double = %d bytes\n", sizeof(double));
```

```
    printf("c = %d bytes\n", sizeof(c));
    printf("i = %d bytes\n", sizeof(i));
    printf("f = %d bytes\n", sizeof(f));
    printf("d = %d bytes\n", sizeof(d));
```

```
    return 0;
```

```
}
```



```
C:\Windows\system32\cmd.exe
Char = 1 bytes
Int = 4 bytes
Float = 4 bytes
Double = 8 bytes
c = 1 bytes
i = 4 bytes
f = 4 bytes
d = 8 bytes
Πιέστε ένα πλήκτρο για συνέχεια. . .
```


Ο τύπος `enum` (I)

- Ο τύπος απαρίθμησης `enum` (*enumeration type*) χρησιμοποιείται για να οριστεί ένα σύνολο ακεραίων με συγκεκριμένα ονόματα και σταθερές τιμές
- Στην απλή περίπτωση δηλώνεται ως εξής:

```
enum όνομα {λίστα απαρίθμησης};
```

- Το αναγνωριστικό όνομα είναι προαιρετικό και δηλώνει το όνομα της απαρίθμησης, π.χ. η εντολή:

```
enum seasons {AUTUMN, WINTER, SPRING, SUMMER};
```

δηλώνει τον τύπο απαρίθμησης `seasons` και τις ακεραίες σταθερές `AUTUMN`, `WINTER`, `SPRING` και `SUMMER`

Ο τύπος `enum` (II)

- Εξ'ορισμού, κατά τη δήλωση ενός τύπου απαρίθμησης, η τιμή της πρώτης σταθεράς αρχικοποιείται με 0
- Αν σε κάποια σταθερά δεν αποδίδεται τιμή, η τιμή της γίνεται ίση με την τιμή της προηγούμενης σταθεράς αυξημένη κατά ένα
- Επομένως, στο προηγούμενο παράδειγμα που δεν αποδίδονται τιμές στις σταθερές, οι τιμές των σταθερών `AUTUMN`, `WINTER`, `SPRING` και `SUMMER` γίνονται 0, 1, 2 και 3, αντίστοιχα
- Σε περίπτωση που στο προηγούμενο παράδειγμα θα θέλαμε να αποδώσουμε συγκεκριμένες τιμές στις σταθερές, θα μπορούσαμε να δηλώσουμε τον τύπο απαρίθμησης π.χ. ως εξής:

```
enum seasons {AUTUMN=10, WINTER, SPRING=30, SUMMER};
```

- Στο παράδειγμα αυτό οι τιμές των σταθερών `AUTUMN`, `WINTER`, `SPRING` και `SUMMER` γίνονται 10, 11, 30 και 31, αντίστοιχα

Δήλωση μεταβλητής τύπου `enum`

- Για να δηλώσουμε μία μεταβλητή σύμφωνα με έναν ήδη δηλωμένο τύπο απαρίθμησης γράφουμε:

```
enum όνομα_τύπου λίστα_μεταβλητών;
```

- Π.χ. με την εντολή:

```
enum seasons s1, s2;
```

δηλώνουμε τις `s1` και `s2` σαν μεταβλητές απαρίθμησης του τύπου `seasons` του προηγούμενου παραδείγματος

- Εναλλακτικά, μπορούμε να δηλώσουμε τις μεταβλητές μαζί με τη δήλωση του τύπου απαρίθμησης, π.χ.:

```
enum seasons {AUTUMN, WINTER, SPRING, SUMMER} s1, s2;
```

Παρατηρήσεις

- Θυμηθείτε ότι η οδηγία `#define` αποτελεί έναν εναλλακτικό τρόπο δήλωσης συμβολικών ονομάτων που αντιστοιχούν σε συγκεκριμένες τιμές
- Η κύρια διαφορά τους είναι ότι ο τύπος `enum` ομαδοποιεί τις σταθερές, ώστε να φαίνεται ότι χαρακτηρίζουν ένα σύνολο τιμών
- Αυτό που πρέπει να θυμάστε με τις μεταβλητές τύπου απαρίθμησης (`enum`) είναι ότι η γλώσσα προγραμματισμού C τις χειρίζεται σαν **ακέραιες μεταβλητές** ενώ τα ονόματα της λίστας απαρίθμησης σαν **ακέραιες σταθερές**

```
enum seasons {AUTUMN, WINTER, SPRING, SUMMER} s1, s2;
```

Οι τελεστές bit

- Οι τελεστές bit χρησιμοποιούνται για το χειρισμό των bits μίας ακέραιας μεταβλητής ή σταθεράς
- Η τιμή ενός bit - ως γνωστόν - μπορεί να είναι 0 ή 1
- Ο υπολογισμός της τιμής μίας έκφρασης που περιέχει τελεστές bit γίνεται με την εφαρμογή τους στα αντίστοιχα bits των τελεστών
- Οι τελεστές bit είναι οι εξής:
 - ◆ Ο τελεστής **AND** &
 - ◆ Ο τελεστής **OR** |
 - ◆ Ο τελεστής **XOR** ^
 - ◆ Ο τελεστής **NOT** ~


Tip Όταν εκτελείτε πράξεις με τελεστές bit είναι ασφαλέστερο να δηλώνετε ως **unsigned** τις αντίστοιχες μεταβλητές αλλιώς, να λαμβάνετε υπόψη σας το bit προσήμου, όταν κάνετε τους υπολογισμούς σας

Ο τελεστής &

- Ο τελεστής & εφαρμόζει τη λογική πράξη **AND** (λογική πράξη **ΚΑΙ**) στα bits των δύο τελεστών και θέτει το bit εξόδου στο 1 μόνο αν τα αντίστοιχα bits και στους δύο τελεστές είναι 1, αλλιώς, το bit εξόδου τίθεται στο 0
- Π.χ. το αποτέλεσμα της πράξης 19 & 2 είναι 2

```
      00010011 (19)
&     00000010 (2)
-----
      00000010 (2)
```

- Ο τελεστής & χρησιμοποιείται συχνά για να μηδενίσει, δηλ. να θέσει ίσα με το μηδέν (0), μια σειρά από bits
 - ♦ Π.χ., η εντολή: `a = a & 3;` κάνει όλα τα bits της `a` εκτός από τα δύο «λιγότερα σημαντικά» bits


 **Μην συγχέετε** τους τελεστές `&&` και `&` (π.χ., αν `a=1` και `b=2`, το αποτέλεσμα της έκφρασης `a && b` είναι 1, ενώ το αποτέλεσμα της έκφρασης `a & b` είναι 0)

Ο τελεστής |

- Ο τελεστής | εφαρμόζει τη λογική πράξη **OR** (λογική πράξη **Ή**) στα bits των δύο τελεστέων και θέτει το bit εξόδου στο 0 μόνο αν τα αντίστοιχα bits και στους δύο τελεστέους είναι 0, αλλιώς, το bit εξόδου τίθεται στο 1
- Π.χ. το αποτέλεσμα της πράξης $19 | 6$ είναι 23

```
      00010011 (19)
|      00000110 (6)
-----
      00010111 (23)
```

- Ο τελεστής | χρησιμοποιείται συχνά για να θέσει ίσα με το ένα (1) μια σειρά από bits
 - ♦ Π.χ., η εντολή: $a = a | 3$; κάνει τα δύο «λιγότερα σημαντικά» bits της μεταβλητής a ίσα με ένα

 **Μην συγχέετε** τους τελεστές `||` και `|` (π.χ., αν $a=1$ και $b=2$, το αποτέλεσμα της έκφρασης $a || b$ είναι 1, ενώ το αποτέλεσμα της έκφρασης $a | b$ είναι 3)

Ο τελεστής \wedge

- Ο τελεστής \wedge εφαρμόζει τη λογική πράξη **XOR** (e**X**clusive **OR**, αποκλειστικό **Ή**) στα bits των δύο τελεστών και θέτει το bit εξόδου στο 1 μόνο αν τα αντίστοιχα bits και στους δύο τελεστές είναι διαφορετικά μεταξύ των, αλλιώς, το bit τίθεται στο 0
- Π.χ. το αποτέλεσμα της πράξης $19 \wedge 6$ είναι 21

	00010011	(19)
\wedge	00000110	(6)

	00010101	(21)

Ο τελεστής ~

- Ο τελεστής συμπληρώματος ~ είναι μοναδιαίος, δηλαδή εφαρμόζεται σε έναν τελεστέο και εφαρμόζει τη λογική πράξη **NOT** (λογική πράξη **ΔΕΝ**)
- Συγκεκριμένα, αντιστρέφει κάθε bit στον τελεστέο του, αλλάζοντας όλα τα 0 σε 1, και αντιστρόφως
- Π.χ. σε ένα 32-bit σύστημα το αποτέλεσμα της πράξης ~ 19 είναι $(2^{32} - 1) - 19$

```
~ 00000000 00000000 00000000 00010011 (19) =  
  11111111 11111111 11111111 11101100
```

Οι τελεστές ολίσθησης

- Οι τελεστές ολίσθησης (\gg και \ll) μετατοπίζουν τα bits μίας ακέραιας μεταβλητής ή σταθεράς κατά ένα συγκεκριμένο αριθμό θέσεων, όπως δείχνουν τα «νοητά βέλη»
- Ο τελεστής \gg μετατοπίζει τα bits της μεταβλητής προς τα δεξιά, όπως δηλαδή δείχνουν τα «νοητά βέλη»
- Ο τελεστής \ll μετατοπίζει τα bits της μεταβλητής προς τα αριστερά, όπως δηλαδή δείχνουν τα «νοητά βέλη»



Επειδή η εφαρμογή των τελεστών ολίσθησης σε αρνητικούς αριθμούς εξαρτάται από τον μεταγλωττιστή, είναι ασφαλέστερο να τους χρησιμοποιείτε σε θετικούς ακεραίους ή unsigned μεταβλητές

Ο τελεστής >>

- Η έκφραση $i \gg n$ μετατοπίζει τα bits της μεταβλητής i κατά n θέσεις δεξιά και τοποθετεί μηδενικά στα n υψηλότερης τάξης bits της μεταβλητής
- Π.χ. ποια θα είναι η τιμή της μεταβλητής a κατά την εκτέλεση του παρακάτω κώδικα;

```
unsigned int a, b = 35;  
a = b >> 2;
```

```
a = 8 , διότι: 00100011 >> 2 = 00001000
```

- Συγκεκριμένα, χάθηκαν τα τελευταία bits 1 και 1 του αρχικού αριθμού (35) και τοποθετήθηκαν τα bits 0 και 0 στην έβδομη και όγδοη θέση, αντίστοιχα
- Και ποια είναι η τιμή της μεταβλητής b ???

Ο τελεστής <<

- Η έκφραση $i \ll n$ μετατοπίζει τα bits της μεταβλητής i κατά n θέσεις αριστερά και τοποθετεί μηδενικά στα n χαμηλότερης τάξης bits της μεταβλητής
- Π.χ. ποια θα είναι η τιμή της μεταβλητής a κατά την εκτέλεση του παρακάτω κώδικα;

```
unsigned int a, b = 35;  
a = b << 2;
```

```
a = 140 , διότι: 00100011 << 2 = 0010001100
```

- Συγκεκριμένα, τα bits του αρχικού αριθμού (35) ολίσθησαν δύο θέσεις αριστερά και τοποθετήθηκαν τα bits 0 και 0 στην πρώτη και στη δεύτερη θέση, αντίστοιχα
- Και ποια είναι η τιμή της μεταβλητής b ???

Παρατηρήσεις (I)

- Όταν χρησιμοποιείται ο τελεστής << και το αποτέλεσμα αποθηκεύεται σε μία μεταβλητή, ο τύπος δεδομένων της μεταβλητής πρέπει να είναι τέτοιος ώστε να μπορεί να αποθηκευτεί η τελική τιμή
- Π.χ. Ποια είναι η έξοδος του προγράμματος ???

```
#include <stdio.h>
int main(void)
{
    char a = 1;

    a <<= 8; /* Ισοδύναμη με a = a << 8; */
    printf("Value = %d\n",a);
    return 0;
}
```

Περιμένετε να τυπωθεί: Value = 256
αλλά τυπώθηκε Value = 0
Γιατί???

Παρατηρήσεις (II)

- Επειδή η θέση ενός bit αντιστοιχεί σε μία δύναμη του 2:
 - ♦ η ολίσθηση ενός θετικού ακεραίου κατά n θέσεις δεξιά (\gg) ισοδυναμεί με τη **διαίρεση** της τιμής του με 2^n
 - Π.χ. Θυμηθείτε $35 \gg 2 = 8$
 - ♦ η ολίσθηση ενός θετικού ακεραίου n θέσεις αριστερά (\ll) ισοδυναμεί με τον **πολλαπλασιασμό** της τιμής του με 2^n
 - Π.χ. Θυμηθείτε $35 \ll 2 = 140$



Na θυμάστε ότι είναι ασφαλέστερο να χρησιμοποιείτε τους τελεστές bit σε **unsigned** μεταβλητές

Προτεραιότητα Τελεστών

- Κάθε τελεστής χαρακτηρίζεται από μία **προτεραιότητα**
- Σε μία έκφραση που περιέχονται περισσότεροι του ενός τελεστές, οι πράξεις εκτελούνται σύμφωνα **με τη σειρά προτεραιότητας** του κάθε τελεστή

- Π.χ. το αποτέλεσμα της πράξης:

$$7 + 5 * 3 - 1 \text{ είναι } 21,$$

γιατί ο τελεστής $*$ έχει μεγαλύτερη προτεραιότητα από τους τελεστές $+$ και $-$, οπότε πρώτα εκτελείται η πράξη $5*3 = 15$ και όχι οι πράξεις $7+5$ ή $3-1$

- Αν μία έκφραση περιέχει διαδοχικούς τελεστές με την ίδια προτεραιότητα, τότε οι πράξεις εκτελούνται σύμφωνα **με τη συσχέτισή τους** (associativity), δηλαδή από αριστερά προς τα δεξιά ή αντίστροφα

- Π.χ. το αποτέλεσμα της πράξης:

$$7 * 4 / 2 * 5 \text{ είναι } 70,$$

γιατί, αφού οι τελεστές $*$ και $/$ έχουν την ίδια προτεραιότητα και συσχέτιση από αριστερά προς τα δεξιά, τότε:

πρώτα εκτελείται ο πολλαπλασιασμός $7*4 = 28$, μετά η διαίρεση $28/2 = 14$ και μετά ο πολλαπλασιασμός $14*5 = 70$

Πίνακας Προτεραιοτήτων

Θέση	Τελεστές	Συσχέτιση
1	() (παρενθέσεις-κλήση συνάρτησης) [] -> . ++ (επιθεματική αύξηση) -- (επιθεματική μείωση)	αριστερά προς δεξιά
2	++ (προθεματική αύξηση) -- (προθεματική μείωση) ! ~ * (έμμεση αναφορά) & (διεύθυνση) + (μοναδιαίο +) - (μοναδιαίο -) (προσαρμογή τύπου) sizeof	δεξιά προς αριστερά
3	* (πολλαπλασιασμός) / %	αριστερά προς δεξιά
4	+ (πρόσθεση) - (αφαίρεση)	αριστερά προς δεξιά
5	<< >>	αριστερά προς δεξιά
6	< <= > >=	αριστερά προς δεξιά
7	== !=	αριστερά προς δεξιά
8	&	αριστερά προς δεξιά
9	^	αριστερά προς δεξιά
10	 	αριστερά προς δεξιά
11	&&	αριστερά προς δεξιά
12	 	αριστερά προς δεξιά
13	?:	δεξιά προς αριστερά
14	= += -= *= /= %= &= ^= = <<= >>=	δεξιά προς αριστερά
15	,	αριστερά προς δεξιά

Παρατηρήσεις

- Όπως φαίνεται στον πίνακα προτεραιοτήτων, κάθε τελεστής χαρακτηρίζεται από μία προτεραιότητα
- Αν μία έκφραση περιέχει διαδοχικούς τελεστές με την ίδια προτεραιότητα, τότε οι πράξεις εκτελούνται σύμφωνα με την συσχέτισή τους
- Προτείνεται η χρήση παρενθέσεων (), ακόμα και όταν δεν χρειάζονται, έτσι ώστε ο κώδικας να είναι πιο ευανάγνωστος και να γίνεται σαφέστερη η σειρά εκτέλεσης των πράξεων
- Είναι πιο σαφές να γράψουμε:
π.χ.1) `a = 7+(5*3)-1;` αντί `a = 7+5*3-1;`
- π.χ.2) `if ((a >> 2) == 10)` αντί `if (a >> 2 == 10)`
- Οι τελεστές `?:`, `[]`, `->`, `.`, `&`, `*` θα παρουσιαστούν σε επόμενες διαλέξεις (έλεγχος προγράμματος/πίνακες/δείκτες/δομές)