Προγραμματισμός I (ECE115)

#3 τελεστές

Χρήστος Αντωνόπουλος
cda@uth.gr

Τελεστής ανάθεσης

- Το σύμβολο της ανάθεσης είναι το **=**
  - **προσοχή: το σύμβολο ελέγχου ισότητας είναι το ==**
- Η μορφή των προτάσεων

  `<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
  - χρησιμοποείται η τρέχουσα τιμή της κάθε μεταβλητής
- Στο δεξί μέρος μπορεί να εμφανίζεται η ίδια μεταβλητή που εμφανίζεται και στο αριστερό
  - η μεταβλητή θα λάβει (κανονικά) την τιμή της έκφρασης

Η διπλή προσωπικότητα της ανάθεσης

- Η έκφραση ανάθεσης αποτελεί (η ίδια) ταυτόχρονα μια **έκφραση αποτίμησης** που **επιστρέφει τιμή**!
- Η τιμή που επιστρέφεται είναι η τιμή που **ανατίθεται** στην μεταβλητή που βρίσκεται στα αριστερά της έκφρασης, δηλαδή η τιμή που παράγεται από την έκφραση στα δεξιά

- 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 = i+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 */
```

Ιδιωματισμοί

- Πολλές φορές χρησιμοποιούμε την ανάθεση για να αλλάξουμε την τιμή μιας μεταβλητής **σε σχέση με την προηγούμενη τιμή της**
- Για το πετύχουμε αυτό, γράφουμε

  `<όνομα> = <όνομα><op><έκφραση>`

  όπου `<op>` ένας τελεστής
- Το ίδιο αποτέλεσμα μπορεί να επιτευχθεί με χρήση του τελεστή `<op>=` της **«σχετικής» ανάθεσης**

  `<όνομα> <op>= <έκφραση>`
- **Προσοχή** στις προτεραιότητες: η έκφραση που εμφανίζεται στα δεξιά αποτιμάται **προτού** εφαρμοστεί ο τελεστής `<op>`

```
int i,j,k;

i = 1;

j = i+1;            /* j γίνεται 2 */

i += 2;             /* i γίνεται 3 */

j *= i+1;           /* j γίνεται 8 */

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

Ιδιωματισμοί αυξομείωσης κατά 1

- Υποστηρίζεται μέσω των τελεστών ++ ή --

  <όνομα>++ ή <όνομα>--

  ++<όνομα> ή --<όνομα>

- **Προσοχή στην τιμή που επιστρέφεται ως αποτέλεσμα της έκφρασης της ανάθεσης!**

- Όταν ο τελεστής εμφανίζεται **πριν** το όνομα της μεταβλητής, επιστρέφεται η **νέα τιμή** της μεταβλητής
- Όταν ο τελεστής εμφανίζεται **μετά** το όνομα της μεταβλητής, 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 ...

- Τα φαινόμενα, μερικές φορές, **απατούν!**

```
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 `int` / `float` / `double`

- **+** : πρόσθεση δύο τιμών
- **–** : αφαίρεση δύο τιμών
- ***** : πολλαπλασιασμός δύο τιμών
- **/** : διαίρεση δύο τιμών
- **%** : υπόλοιπο διαίρεσης δύο **ακέραιων** τιμών
  - `x = y*(x/y) + x%y`, όμως **δεν** ισχύει πάντα η «μαθηματική» ιδιότητα του υπολοίπου (>=0), π.χ. όταν ο αριθμητής είναι αρνητικός: `-5%4` ή `-5%-4`

- Η αποτίμηση γίνεται **από αριστερά προς τα δεξιά**
- Τα '*', '/' και '%' έχουν **υψηλότερη προτεραιότητα** από τα '+' και '–'

Αριθμητικοί τελεστές `int` / `float` / `double`

- Ο τύπος του αποτελέσματος εξαρτάται από τον τύπο των τιμών.
  - Αν έστω και μία τιμή είναι δεκαδικός αριθμός, το αποτέλεσμα είναι δεκαδικός
    - `3.1 + 2` -> `5.1`
    - `9.0 / 3` -> `3.0`
  - Αν και οι δύο τιμές είναι ακέραιοι, το αποτέλεσμα είναι ακέραιος.
    - **ΠΡΟΣΟΧΗ** στην ακέραια διαίρεση!
    - `8 / 3` -> `2`
    - `4 / 10` -> `0`
    - Θα δούμε αργότερα πώς μπορούμε να παράγουμε δεκαδικό αριθμό σε αυτές τις περιπτώσεις.

Δυαδικοί τελεστές σε επίπεδο bits

- **`&`** : δυαδικό «and» (`0&0->0, 0&1->0, 1&0->0, 1&1->1`)
- **`|`** : δυαδικό «or» (`0|0->0, 0|1->1, 1|0->1, 1|1->1`)
- **`^`** : δυαδικό «xor» (`0^0->0, 0^1->1, 1^0->1, 1^1->0`)
- **`~`** : δυαδικό «not» (`~0->1, ~1->0`)

- **`<<`** : αριστερή ολίσθηση
  - τα νέα (λιγότερο σημαντικά) bits παίρνουν **πάντα** την τιμή 0
- **`>>`** : δεξιά ολίσθηση
  - τα νέα (περισσότερο σημαντικά) bits παίρνουν **αναλόγως**
    - την όποια τιμή έχει το πιο σημαντικό bit του ορίσματος, αν αυτό ερμηνεύεται ως signed (**arithmetic shift**)
    - 0 αν το όρισμα ερμηνεύεται ως unsigned (**logical shift**)

Παρένθεση: δεκαεξαδική αναπαράσταση

- Πως συμβολίζουμε την τιμή ενός byte;
- Δίνουμε τιμές (`0` ή `1`) για κάθε ένα από τα 8 bits;
- Δεν είναι πρακτικό … για τον άνθρωπο

Βολική λύση: **δεκαεξαδικό σύστημα**

- Χρειαζόμαστε ψηφία για να αναπαρασταθούν όλες οι τιμές από το 0 έως το 15
  - `0-9, A(10), B(11), C(12), D(13), E(14), F(15)`
- 1 byte μπορεί να αναπαρασταθεί μόνο με 2 ψηφία, ένα ψηφίο για τα 4 λιγότερο σημαντικά bits και ένα ψηφίο τα 4 πιο σημαντικά bits
  - `00000000` -> `00`, `11111111` -> `FF`, `10100111` -> `A7`

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

a = 0x61;      /* a γίνεται   01100001 */

b = 0x62;      /* b γίνεται   01100010 */

c = a|b;       /* c γίνεται   01100011, x63 */

c = a&b;       /* c γίνεται   01100000, x60  */

c = a^b;       /* c γίνεται   00000011, x03 */

d = c = ~a;    /* d,c γίνονται 10011110, x9E */

d = d>>3;      /* d γίνεται   000 10011, x13  */

c = c>>3;      /* c γίνεται   111 10011, xF3 */
```

Γρήγορος πολλαπλασιασμός

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

```
v = v<<i; /* v = v*2^i */
v = v>>i; /* v = v/2^i */
```

- Επίσης, μπορούμε να παράγουμε πολύ εύκολα (και γρήγορα) τιμές που είναι δυνάμεις του 2

```
v = 1<<i; /* v = 2^i */
```

```
short int i;

i = 5;          /* i γίνεται 00000000 00000101 (5) */


i = i<<4;       /* i γίνεται 00000000 01010000 (5x2^4=80) */

i = i>>2;       /* i γίνεται 00000000 00010100 (80/2^2=20) */


i = 1<<8;       /* i γίνεται 00000001 00000000 (1x2^8=256) */
```

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)
- Το αποτέλεσμα είναι `0` ή `1` (για ψευδές ή αληθές)
  - 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 σύνδεσμοι

- `||` : λογικό «ή»
- `&&` : λογικό «και»
- 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)
  - στο `||` η αποτίμηση σταματά μόλις προκύψει «αληθές»
    - το αποτέλεσμα είναι αναγκαστικά «αληθές» (`1`)
  - στο `&&` η αποτίμηση σταματά μόλις προκύψει «ψευδές»
    - το αποτέλεσμα είναι αναγκαστικά «ψευδές» (`0`)
- **Προσοχή στις παρενέργειες!**

Παρενέργεια

- Αλλαγή (ή όχι) τιμής μιας μεταβλητής **χωρίς** αυτό να είναι εύκολα ορατό από τον κώδικα

- Π.χ. `<lexpr> && (a++)` αλλάζει την τιμή της `a` **μόνο όταν** η `<lexpr>` αποτιμάται ως αληθής
- Π.χ. `<lexpr> || (a++)` αλλάζει την τιμή της `a` **μόνο όταν** η `<lexpr>` αποτιμάται ως ψευδής

- Ο προγραμματισμός με παρενέργειες θεωρείται κακό στυλ γιατί οδηγεί σε **δυσνόητο** κώδικα
  - και συχνά σε bugs που δύσκολα εντοπίζονται

 σημείο όπου σταματά η αποτίμηση της έκφρασης

```
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 μένει 2 */

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

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

Υπόλοιποι 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, δηλαδή της <exprn>

```
int a = 1, b = 2, c;
c = (a<b)? a+b : b;        /* c γίνεται 3 */
c = (a*=2, a+=1, b=0); /* a,b,c γίνονται 3,0,0 */
c = a*=2, a+=1, b=0;   /* a,b,c γίνονται 7,0,6 */
```

Προτεραιότητες

