Προγραμματισμός Υπολογιστών στην C++

Τελεστές
Έλεγχος Ροής Προγράμματος

Ο Τελεστής Εκχώρησης =

- Ο τελεστής = χρησιμοποιείται για την εκχώρηση τιμής σε μία μεταβλητή. Π.χ. με την εντολή: `a = 10;` η τιμή του `a` γίνεται ίση με `10`, ενώ με την εντολή `a = b;` η τιμή του `a` γίνεται ίση με την τιμή του `b`

- Αν χρησιμοποιηθεί διαδοχικά 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: `a = b = c = 10;` οι τιμές των μεταβλητών `a`, `b` και `c` γίνονται ίσες με `10`

- Αν ο τύπος της μεταβλητής και της εκχωρούμενης τιμής δεν είναι ίδιος, τότε, η τιμή πρώτα μετατρέπεται στον τύπο της μεταβλητής, εφόσον αυτό είναι δυνατό, και μετά εκχωρείται σε αυτήν. Π.χ.

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

Ποιες τιμές αποθηκεύονται στις μεταβλητές `a` και `b`;

Παρατηρήσεις

- Ο δεξιός τελεστέος είναι μία έκφραση που έχει τιμή, όπως μία μεταβλητή ή μία σταθερά. Ο αριστερός τελεστέος πρέπει να είναι μία *lvalue (left value – αριστερή τιμή)*. Μία *lvalue* αναφέρεται σε μία οντότητα που είναι αποθηκευμένη στη μνήμη (π.χ. μεταβλητή) στην οποία μπορούμε να αποθηκεύσουμε κάτι. Π.χ, δεν επιτρέπεται να είναι σταθερά ή το αποτέλεσμα ενός υπολογισμού:

```
20 = a; // Λάθος
b-a = 30; // Λάθος
```

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

- Επίσης, η *lvalue* πρέπει να είναι τροποποιήσιμη. Για παράδειγμα, θα δούμε στη συνέχεια ότι το όνομα πίνακα δεν είναι μία τροποποιήσιμη *lvalue*

- Το συμπληρωματικό της *lvalue* έννοιας είναι η *rvalue (right value – δεξιά τιμή)*. Σαν γενικός κανόνας, μια *rvalue* είναι μια έκφραση που δεν είναι *lvalue*, όπως μια σταθερή τιμή (π.χ. 10) ή μία έκφραση (π.χ. a+b)

Αριθμητικοί Τελεστές (1/3)

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

- Αν και οι δύο τελεστέοι είναι ακέραιοι, τότε ο τελεστής / αποκόπτει 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 ένας από τους δύο τελεστέους είναι πραγματική σταθερά ή μεταβλητή, 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

Αριθμητικοί Τελεστές (2/3)

- Ο τελεστής % χρησιμοποιείται για τον υπολογισμό **του υπολοίπου της διαίρεσης** δύο ακεραίων τελεστέων. 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`

- **Προσοχή!** Ο τελεστής % μπορεί να εφαρμοστεί μόνο μεταξύ ακεραίων αριθμών, αλλιώς ο μεταγλωττιστής εμφανίζει μήνυμα λάθους

Αριθμητικοί Τελεστές (3/3)

- Οι πέντε αυτοί τελεστές λέγονται και «δυαδικοί» (binary), επειδή απαιτούν δύο τελεστέους
- Η C++ παρέχει και τους μοναδιαίους (unary) τελεστές + και – που απαιτούν μόνο έναν τελεστέο
  - Ο μοναδιαίος τελεστής + δεν έχει κάποια επίδραση στον τελεστέο
  - Το αποτέλεσμα με τον μοναδιαίο – είναι η αντίθετη τιμή του τελεστέου
- Όταν σε κάποια έκφραση περιέχονται και μοναδιαίοι και δυαδικοί τελεστές + και –, ο μεταγλωττιστής αντιλαμβάνεται 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 δυαδικό 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.*/
```

Η μεταβλητή b αποκτά την τιμή 15 και η a την τιμή -10

Ο Τελεστής Αύξησης ++ (1/2)

- Ο τελεστής αύξησης ++ εισάγεται πριν ή μετά από το όνομα του τελεστέου
- Σε κάθε περίπτωση η τιμή του τελεστέου αυξάνεται κατά ένα
- Ο τελεστέος πρέπει να είναι μία τροποποιήσιμη «αριστερή-τιμή» *(lvalue)*, όπως μία μεταβλητή. Π.χ.

```
int a = 10;
a++; // Ισοδύναμη με a = a+1;
++a; // Ισοδύναμη με a = a+1;
(a+10)++; // Λάθος
```

Η τιμή του a αυξάνεται δύο φορές κατά ένα και γίνεται ίση με 12.

Ο Τελεστής Αύξησης ++ (2/2)

- Στην επιθεματική μορφή, η αύξηση πραγματοποιείται αφού πρώτα χρησιμοποιηθεί η τιμή της μεταβλητής στην έκφραση. Π.χ., με τις εντολές:

```
int a = 10, b;
b = a++;
```

πρώτα αποθηκεύεται στη μεταβλητή b η τρέχουσα τιμή του a και μετά η τιμή του a αυξάνεται. Επομένως, το a θα γίνει 11 και το b ίσο με 10

- Στην προθεματική μορφή, πρώτα αυξάνεται η τιμή της μεταβλητής και μετά αυτή χρησιμοποιείται στην έκφραση. Π.χ., με τις εντολές:

```
int a = 10, b;
b = ++a;
```

πρώτα αυξάνεται η τιμή του a και μετά αυτή αποθηκεύεται στο b. Επομένως, και οι δύο μεταβλητές θα γίνουν ίσες με 11

Ο Τελεστής Μείωσης --

- Ο τελεστής μείωσης -- εισάγεται πριν ή μετά από το όνομα μίας μεταβλητής
- Σε κάθε περίπτωση η τιμή της μεταβλητής μειώνεται κατά ένα
- Σε μία έκφραση, ο τελεστής -- συμπεριφέρεται όπως και ο τελεστής ++. Π.χ.

```
double a = 1.23;
cout << ++a << '\n';
cout << a-- << '\n';
```

Πρώτα η τιμή του a αυξάνεται και ο κώδικας εμφανίζει 2.23. Μετά, πρώτα ο κώδικας εμφανίζει την τιμή του a, δηλαδή 2.23, και μετά αυτή μειώνεται

Παρατηρήσεις

- Αν εφαρμόσετε τους τελεστές ++ και -- για να αλλάξετε την τιμή μιας μεταβλητής σε μία έκφραση, μην χρησιμοποιήσετε ξανά τη μεταβλητή στην ίδια έκφραση, επειδή η σειρά αποτίμησης είναι απροσδιόριστη

- Π.χ., το αποτέλεσμα της έκφρασης:

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

είναι απροσδιόριστο, εξαρτάται από τον μεταγλωττιστή αν πρώτα αυξήσει την τιμή του a και μετά κάνει τον πολ/σμό, ή όχι.

Σχεσιακοί Τελεστές (1/2)

- Οι σχεσιακοί τελεστές >, >=, <, <=, !=, ==, χρησιμοποιούνται για τη σύγκριση των τιμών δύο τελεστέων

- Συνήθως χρησιμοποιούνται σε εντολές ελέγχου (π.χ. στην εντολή `if`) και σε επαναληπτικούς βρόχους (π.χ. στην εντολή `for`)

- Π.χ.
  `if(a > 5)` ελέγχουμε αν η τιμή του a είναι μεγαλύτερη από το 5.
  `if(a >= 5)` ελέγχουμε αν η τιμή του a είναι μεγαλύτερη ή ίση από το 5.
  `if(a < 5)` ελέγχουμε αν η τιμή του a είναι μικρότερη από το 5.
  `if(a <= 5)` ελέγχουμε αν η τιμή του a είναι μικρότερη ή ίση από το 5.
  `if(a != 5)` ελέγχουμε αν η τιμή του a είναι διαφορετική από το 5.
  `if(a == 5)` ελέγχουμε αν η τιμή του a είναι ίση με το 5.

Σχεσιακοί Τελεστές (2/2)

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

- Το αποτέλεσμα που παράγουν οι σχεσιακοί τελεστές είναι τύπου `bool`, δηλαδή, `true` ή `false`. Π.χ. το αποτέλεσμα της έκφρασης (`a > 10`) είναι `true` μόνο αν η τιμή της μεταβλητής `a` είναι μεγαλύτερη από το `10`, αλλιώς είναι `false`

- Ποια είναι η έξοδος του διπλανού προγράμματος:

```
#include <iostream>
int main()
{
        int a = 4, b = 6, c;

        a = (a <= (b-2)) + (b > (a+1));
        b = (a == 2) > ((b-3) < 3);
        c = (b != 0);
        std::cout << a << ' ' << b << ' ' << c << '\n';
        return 0;
}
```

Σχεσιακοί Τελεστές (2/2)

```
Έξοδος: 2  1  1
```

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 την εκχώρηση τιμής
- Π.χ., η τιμή της έκφρασης:

```
a == 5
```

είναι `true`, μόνο η τιμή της μεταβλητής `a` είναι `5`, αλλιώς είναι `false`

ενώ η τιμή της έκφρασης:

```
a = 5
```

εκχωρεί την τιμή `5` στο `a`

Συνδυαστικοί Τελεστές

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

```
exp1 op= exp2;
```

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

```
exp1 = exp1 op (exp2);
```

- 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 <iostream>
int main()
{
    int a = 1, b = 2;

    a -= 2;
    a *= 1-b;
    a += b+3;
    a /= b+2;
    a %= b;
    std::cout << a << '\n';
    return 0;
}
```

Παράδειγμα

```
Έξοδος: 1
```

Λογικοί Τελεστές (1)

- **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 **false**
  - Αν μία έκφραση **exp** είναι **ψευδής** (δηλαδή έχει **μηδενική τιμή**), τότε το αποτέλεσμα της πράξης **!exp** είναι **true**
  - 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)`

Λογικοί Τελεστές (2)

- **Ο τελεστής &&**
  - Ο τελεστής && εκτελεί τη λογική πράξη AND μεταξύ δύο τελεστέων
  - Η τιμή της έκφρασης είναι **true**, μόνο αν και οι δύο τελεστέοι είναι αληθείς. Αλλιώς, η τιμή της είναι **false**
  - Αν ο όρος που αποτιμάται σε μία έκφραση με τον τελεστή && είναι ψευδής, ο μεταγλωττιστής δεν αποτιμά τους επόμενους όρους *(short circuit evaluation)* και θέτει κατευθείαν την τιμή της συνολικής έκφρασης ίση με **false**. Π.χ. στο επόμενο πρόγραμμα η τιμή του b δεν αυξάνεται, γιατί ο πρώτος όρος (a < 2) έχει ψευδή τιμή:

```
#include <iostream>
int main()
{
    int a = 5, b = 8, c;
    c = (a < 2) && (++b > 3);
    std::cout << c << ' ' << b << '\n';
    return 0;
}
```

Επομένως, το πρόγραμμα εμφανίζει: 0 8

Λογικοί Τελεστές (3)

- **Ο τελεστής ||**
  - Ο τελεστής || εκτελεί τη λογική πράξη OR μεταξύ δύο τελεστέων
  - Η τιμή της έκφρασης είναι `true`, αν έστω και ένας από τους δύο τελεστέους είναι αληθής. Αν και οι δύο τελεστέοι είναι ψευδείς, η τιμή της έκφρασης είναι `false`
  - Όπως και ο τελεστής `&&`, ο τελεστής || εφαρμόζει *short circuit* αποτίμηση. Αν ο όρος που αποτιμάται είναι αληθής, ο μεταγλωττιστής δεν αποτιμά τους επόμενους όρους και θέτει κατευθείαν την τιμή της συνολικής έκφρασης ίση με `true`. Π.χ., στο επόμενο πρόγραμμα η τιμή του `b` δεν αυξάνεται, γιατί ο πρώτος όρος (`a > 2`) είναι αληθής:

```
#include <iostream>
int main()
{
    int a = 5, b = 8, c;
    c = (a > 2) || (++b > 3);
    std::cout << c << ' ' << b << '\n';
    return 0;
}
```

Επομένως, το πρόγραμμα εμφανίζει: `1 8`

Ο Τελεστής κόμμα ,

- Ο τελεστής κόμμα (,) χρησιμοποιείται για τη σύνδεση εκφράσεων, ώστε να σχηματιστεί μία μοναδική έκφραση:

```
exp1, exp2, exp3,...
```

- Επειδή ο τελεστής κόμμα (,) έχει αριστερή συσχέτιση, πρώτη εκτελείται η **exp1**, μετά η **exp2**, μετά η **exp3**, έως και την τελευταία έκφραση. Π.χ., αφού οι εκφράσεις εκτελούνται από αριστερά προς τα δεξιά ο παρακάτω κώδικας θα εμφανίσει 30:

```
int b;
b = 10, b += 20, cout << b;
```

- Ο τύπος και η τιμή της συνολικής έκφρασης είναι ίδιοι με τον τύπο και την τιμή της τελευταίας έκφρασης

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος?

```
#include <iostream>
int main()
{
    int a, b, c;

    a = (b = 30, b = b/8, ++b);
    std::cout << a << ' ' << b << '\n';
    c = (b != 4);
    if(a, b, c)
          std::cout << "One\n";

    return 0;
}
```

Παράδειγμα

- Απάντηση. Με τις δύο πρώτες εκφράσεις η τιμή του b γίνεται 3. Με την έκφραση ++b πρώτα θα αυξηθεί η τιμή του b και μετά αυτή θα αποθηκευτεί στο a. Άρα, το πρόγραμμα θα εμφανίσει: 4 4. Αφού το b είναι 4, η συνθήκη είναι ψευδής και το c γίνεται 0. Στη συνέχεια, επειδή η τιμή της συνολικής έκφρασης είναι η τιμή του c και επειδή αυτή είναι 0, το πρόγραμμα δε θα εμφανίσει τίποτα.

Παρατηρήσεις

- Επειδή η χρήση του τελεστή κόμμα μπορεί να παράξει δυσνόητο και δυσκολότερο να ελεγχθεί κώδικα, δεν χρησιμοποιείται συχνά

- Η συνηθέστερη χρήση του είναι 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 το κόμμα που χρησιμοποιούμε για να ξεχωρίζουν μεταβλητές σε δηλώσεις, ορίσματα συναρτήσεων και τιμές σε λίστες αρχικοποίησης δεν είναι ο τελεστής, αλλά το απλό διαχωριστικό

Ο Τελεστής sizeof

- Ο τελεστής sizeof υπολογίζει το μέγεθος, σε οκτάδες, κάποιου τύπου δεδομένων, μίας μεταβλητής, σταθεράς ή έκφρασης. Π.χ., το επόμενο πρόγραμμα χρησιμοποιεί τον τελεστή sizeof για να εμφανίσει πόσες οκτάδες δεσμεύουν οι μεταβλητές του προγράμματος:

```
#include <iostream>
int main()
{
    char c;
    int i;
    double d;

    std::cout << sizeof(c) << ' ' << sizeof(i) << ' ' <<
sizeof(d) << '\n';
    return 0;
}
```

Ο Τύπος enum (1)

- Ο τύπος απαρίθμησης **enum** *(enumeration type)* χρησιμοποιείται για να οριστεί ένα σύνολο ακεραίων με συγκεκριμένα ονόματα και σταθερές τιμές
- Τα ονόματα πρέπει να διαφέρουν μεταξύ τους καθώς και με ονόματα μεταβλητών στην ίδια εμβέλεια, αλλά οι τιμές τους δεν χρειάζεται να είναι διαφορετικές

- Συνήθως, 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 ακέραιες σταθερές AUTUMN, WINTER, SPRING και SUMMER

- Θυμηθείτε ότι με την οδηγία **#define** ή με το προσδιοριστικό **const** μπορούμε επίσης να δηλώσουμε σταθερές. Η κύρια διαφορά τους με τον τύπο **enum** είναι ότι ο τύπος **enum** ομαδοποιεί τις σταθερές, ώστε να φαίνεται ότι χαρακτηρίζουν ένα σύνολο τιμών

Ο Τύπος enum (2)

- 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:

```
όνομα_τύπου λίστα_μεταβλητών;
```

- Π.χ. με την εντολή:

```
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;
```

Οι Τελεστές bit

- Οι τελεστές bit χρησιμοποιούνται για το χειρισμό των bits μίας ακέραιας μεταβλητής ή σταθεράς
- Η τιμή ενός bit 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 τελεστέων
- Οι τελεστές bit είναι οι εξής:
  - Ο τελεστής AND &
  - Ο τελεστής OR |
  - Ο τελεστής XOR ^
  - Ο τελεστής NOT ~
- Όταν εκτελείτε πράξεις με τελεστές bit είναι ασφαλέστερο να τους εφαρμόζετε σε `unsigned` 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 τελεστέων και θέτει το bit εξόδου στο 1 μόνο αν τα αντίστοιχα bits και στους δύο τελεστέους 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 0 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 ^

- Ο τελεστής ^ εφαρμόζει τη λογική πράξη XOR (eXclusive OR , αποκλειστικό Ή) στα bits των δύο τελεστέων και θέτει το bit εξόδου στο 1 μόνο αν τα αντίστοιχα bits και στους δύο τελεστέους είναι διαφορετικά μεταξύ των, αλλιώς, το bit τίθεται στο 0

- Π.χ. το αποτέλεσμα της πράξης 19 ^ 6 είναι 21

```
        00010011 (19)
^       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 (>> και <<) μετατοπίζουν τα bits μίας ακέραιας μεταβλητής ή σταθεράς κατά ένα συγκεκριμένο αριθμό θέσεων, όπως δείχνουν τα «νοητά βέλη»

- Ο τελεστής >> μετατοπίζει τα bits της μεταβλητής προς τα δεξιά, όπως δηλαδή δείχνουν τα «νοητά βέλη»

- Ο τελεστής << μετατοπίζει τα bits της μεταβλητής προς τα αριστερά, όπως δηλαδή δείχνουν τα «νοητά βέλη»

Ο Τελεστής >>

- Η έκφραση `i >> n` μετατοπίζει τα bits της μεταβλητής `i` κατά `n` θέσεις δεξιά. Αν η τιμή του `i` είναι θετική ή ο τύπος του είναι unsigned, οι τιμές των n υψηλότερης τάξης bits που εισάγονται στα αριστερά είναι μηδέν
- Επειδή η θέση ενός bit αντιστοιχεί σε μία δύναμη του 2, όταν ένας θετικός ακέραιος ολισθαίνει n θέσεις δεξιά η τιμή του διαιρείται με 2^n
- 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 << n` μετατοπίζει τα bits της μεταβλητής `i` κατά `n` θέσεις αριστερά και τοποθετεί μηδενικά στα `n` χαμηλότερης τάξης bits της μεταβλητής
- Αν δεν συμβεί υπερχείλιση, όταν ένας θετικός ακέραιος ολισθαίνει n θέσεις αριστερά η τιμή του πολ/ζεται με 2^n
- 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

Παρατηρήσεις

- 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 θα είναι η τιμή της μεταβλητής a κατά την εκτέλεση του παρακάτω κώδικα;

```
unsigned char a = 32;
a <<= 3; // Ισοδύναμη με a = a << 3;
```

Παρατηρήσεις

- Απάντηση. Η εντολή `a <<= 3` ολισθαίνει την τιμή του `a` τρεις θέσεις αριστερά και εισάγει τρία μηδενικά bits στα δεξιά. Άρα, το αποτέλεσμα της ολίσθησης είναι 100000000. Όμως, αφού ο τύπος του `a` είναι `unsigned char`, αποθηκεύεται μόνο η τιμή που υπάρχει στα οκτώ τελευταία bits. Άρα, η τιμή του `a` γίνεται `0`

- Τελειώνοντας τη συζήτησή μας για τους τελεστές `<<` και `>>` είναι λογικό να σας έχει δημιουργηθεί η απορία, πώς ακριβώς χρησιμοποιούνται με τα `cout` και `cin` αντικείμενα; Η απάντηση είναι ότι εκεί δε λειτουργούν σαν τελεστές ολίσθησης. Όπως θα μάθουμε αργότερα, η κλάση `ostream` τους υπερφορτώνει και αλλάζει η συμπεριφορά τους. Ανάλογα με την έκφραση, ο μεταγλωττιστής αντιλαμβάνεται τον τρόπο με τον οποίου πρέπει να τους

Εναλλακτικές Αναπαραστάσεις Τελεστών

- Η γλώσσα παρέχει ένα σύνολο δεσμευμένων λέξεων για την εναλλακτική αναπαράσταση των παρακάτω τελεστών:

&&	&=	&	|	~	!	!=	||	|=	^	^=
and	and_eq	bitand	bitor	compl	not	not_eq	or	or_eq	xor	xor_eq

Για παράδειγμα, οι παραστάσεις:

```
bool a = (b && c) || !d;
int a = (b & c) | ~d;
```

είναι ισοδύναμες με:

```
bool a = (b and c) or not d;
int a = (b bitand c) bitor compl d;
```

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` 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` 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`

Παρατηρήσεις

- Δεν χρειάζεται να απομνημονεύσετε την προτεραιότητα και τη συσχέτιση των τελεστών. Όταν δεν είστε σίγουροι για την προτεραιότητα των τελεστών να χρησιμοποιείτε παρενθέσεις
- Επίσης, συστήνεται η χρήση παρενθέσεων `( )` ακόμα και όταν δεν χρειάζονται, έτσι ώστε ο κώδικας να διαβάζεται πιο εύκολα και να γίνεται σαφέστερη στον αναγνώστη η σειρά αποτίμησης των εκφράσεων. Π.χ. είναι πιο σαφές να γράψουμε:

  `a = b-(c/d)+d;` αντί για `a = b-c/d+d;`

Η Εντολή if

- Η εντολή **if** ελέγχει τη ροή ενός προγράμματος ανάλογα με την τιμή μίας συνθήκης
- Στην πιο απλή μορφή της συντάσσεται ως εξής:

```
if(συνθήκη)
{
        ... /* ομάδα εντολών */
}
```

- Αν η συνθήκη είναι **αληθής** (**true**), τότε εκτελούνται οι εντολές που περικλείονται στα άγκιστρα **{...}**
- Αν η συνθήκη είναι **ψευδής** (**false**), τότε το μπλοκ των εντολών που περικλείεται στα άγκιστρα παρακάμπτεται και συνεπώς δεν εκτελείται
- Αν το μπλοκ εντολών περιέχει μόνο μία εντολή, τότε τα άγκιστρα μπορούν να παραλειφθούν
- Η έκφραση: **if(x)** είναι ισοδύναμη με **if(x != 0)**
- Η έκφραση: **if(!x)** είναι ισοδύναμη με **if(x == 0)**

Παρατηρήσεις (1)

- **Προσοχή!** Ένα πολύ συνηθισμένο λάθος είναι να προστίθεται το ; στο τέλος της **if** εντολής, όπως κάνουμε με τις απλές εντολές
- Το ; θεωρείται ξεχωριστή εντολή, η οποία απλά δεν κάνει τίποτα *(null statement)*, με αποτέλεσμα ο μεταγλωττιστής να τερματίζει την εκτέλεση της **if**
- Π.χ. τί εμφανίζει ο παρακάτω κώδικας;

```
int x = 10;
if(x < 0); // Προσοχή.
      cout << "Yes\n";
```

Το ; τερματίζει την εκτέλεση της **if**, άρα η **cout** εντολή δε συνδέεται μαζί της. Επομένως, αυτός ο κώδικας εμφανίζει **Yes** ανεξάρτητα από την τιμή του **x**

Παρατηρήσεις (2)

- **Προσοχή!** Ένα ακόμα συνηθισμένο λάθος είναι να συγχέεται ο τελεστής εκχώρησης = με τον τελεστή ελέγχου ισότητας ==
- Π.χ. τί εμφανίζει ο παρακάτω κώδικας;

```
int x = 10;
if(x = 30)
        cout << "Yes\n";
```

ο κώδικας δεν ελέγχει αν το **x** είναι 30, αλλά εκχωρεί την τιμή 30 στο **x**. Άρα, η συνθήκη είναι αληθής και ο κώδικας εμφανίζει Yes. Παρόμοια, ο παρακάτω κώδικας δεν εμφανίζει τίποτα:

```
int x = 0;
if(x = 0)
        cout << "Yes\n";
```

γιατί με την εκχώρηση του 0 στο **x**, η συνθήκη γίνεται ψευδής

Η Εντολή if-else

- Η εντολή **if** μπορεί προαιρετικά να συμπληρώνεται με την εντολή **else**:

```
if(συνθήκη)
{
        ... /* ομάδα εντολών Α */
}
else
{
        ... /* ομάδα εντολών Β */
}
```

Όταν η **συνθήκη** είναι **αληθής** (**true**), τότε εκτελείται η ομάδα εντολών Α (δηλ. οι εντολές που περιέχονται ανάμεσα στα άγκιστρα του **if**), ενώ όταν η συνθήκη είναι **ψευδής** (**false**), τότε εκτελείται η ομάδα εντολών Β (δηλ. οι εντολές που περιέχονται ανάμεσα στα άγκιστρα του **else**)

Παρατηρήσεις

- **Προσοχή!** Όπως και στην περίπτωση του **if**, ένα συνηθισμένο λάθος είναι να προστεθεί το ; στο τέλος του **else**
- Π.χ. τί εμφανίζει ο παρακάτω κώδικας;

```
int i = 20, j = 10, max;
if(i > j)
        max = i;
else; // Προσοχή.
        max = j;
cout << max;
```

Αρχικά το **max** γίνεται 20, μετά όμως, επειδή το ; τερματίζει την εκτέλεση του **else**, γίνεται 10. Έτσι, ο κώδικας εμφανίζει 10

Ένθετες if Εντολές

- Μία if εντολή μπορεί να περιέχει ένθετες if και else εντολές, οι οποίες με τη σειρά τους μπορεί να περιέχουν και άλλες, κ.ο.κ. Για παράδειγμα, ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int a = -1, b = 0, c = 1;

        if(a < b && b < c)
        {
                if(!b)
                {
                        if(-a == c)
                                std::cout << "One\n";
                        else
                                std::cout << "Two\n";
                }
        }
        else
                std::cout << "Three\n";
        return 0;
}
```

Ένθετες if Εντολές

- Απάντηση. Αφού η συνθήκη στην πρώτη `if` είναι αληθής εκτελείται η ένθετη `if`. Αφού το `b` είναι `0`, η συνθήκη είναι αληθής και εκτελείται η επόμενη ένθετη `if`. Αφού η συνθήκη είναι αληθής το πρόγραμμα εμφανίζει `One`

Ένθετες if Εντολές

- Επειδή η **else** είναι προαιρετική, η παράλειψή της σε μία αλληλουχία από ένθετες **if** εντολές μπορεί να προκαλέσει σύγχυση. Αυτή η περίπτωση συχνά αναφέρεται ως αιωρούμενη (dangling) **else**
- Για να ταιριάζετε **if** και **else** εντολές, ο κανόνας είναι ότι κάθε **else** συνδέεται με την πιο κοντινή **if** που υπάρχει στην ίδια ομάδα εντολών και η οποία δεν σχετίζεται με άλλη **else**. Για παράδειγμα, ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int a = 10;

        if(a != 10)
                if(a < 30)
                        std::cout << "1\n";
        else
                std::cout << "2\n";
        return 0;
}
```

Ένθετες if Εντολές

- Απάντηση. Σύμφωνα με τον παραπάνω κανόνα, η else συνδέεται με την πιο κοντινή if που ανήκει στην ίδια ομάδα εντολών. Αυτή είναι η δεύτερη if. Άρα, αφού η συνθήκη στην πρώτη if είναι ψευδής το πρόγραμμα δεν εμφανίζει τίποτα

- Σημειώστε ότι επίτηδες χρησιμοποίησα αυτή τη στοίχιση για να σας παραπλανήσω και να νομίσετε ότι η else συνδέεται με την πρώτη if. Στοιχίζοντας κάθε else με την αντίστοιχη if, γράφοντας τον κώδικα σε εσοχές και χρησιμοποιώντας άγκιστρα, ακόμα και όταν αυτά δεν χρειάζονται, ο κώδικας διαβάζεται και ελέγχεται πιο εύκολα. Για παράδειγμα, δείτε πόσο πιο εύκολα διαβάζεται και ελέγχεται το προηγούμενο πρόγραμμα:

```
#include <iostream>
int main()
{
        int a = 10;
        if(a != 10)
        {
                if(a < 30)
                        std::cout << "1\n";
                else
                        std::cout << "2\n";
        }
        return 0;
}
```

Έλεγχος Σειράς Συνθηκών

- Όταν θέλουμε να ελέγξουμε μία σειρά από συνθήκες, συνήθως χρησιμοποιείται η ακόλουθη σύνταξη:

```
if(συνθήκη_A)
{
      ... /* ομάδα εντολών A */
}
else if(συνθήκη_B)
{
      ... /* ομάδα εντολών B */
}
else if(συνθήκη_C)
{
      ... /* ομάδα εντολών C */
}
.
.
.
else
{
      ... /* ομάδα εντολών N */
}
... /* επόμενες εντολές του προγράμματος. */
```

- Όταν βρεθεί μία συνθήκη που να είναι αληθής, τότε εκτελείται η ομάδα εντολών που σχετίζεται με αυτή και οι υπόλοιπες `if` συνθήκες αγνοούνται
- Η εκτέλεση του κώδικα συνεχίζει με την πρώτη εντολή που υπάρχει μετά την τελευταία `else` εντολή
- Η τελευταία `else` είναι προαιρετική. Αν υπάρχει, το τμήμα εντολών της εκτελείται μόνο αν όλες οι συνθήκες είναι ψευδείς

Παράδειγμα

- Το παρακάτω πρόγραμμα διαβάζει δύο ακεραίους και εμφανίζει το αποτέλεσμα της σύγκρισής τους:

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
    int i, j;

    cout << "Enter numbers: ";
    cin >> i >> j;
    if(i < j)
        cout << i << " < " << j << '\n';
    else if(i > j)
        cout << i << " > " << j << '\n';
    else
        cout << i << " = " << j << '\n';
    return 0;
}
```

Παράδειγμα

Να συμπληρώσετε τα κενά στο πρόγραμμα (δεν επιτρέπεται να προσθέσετε άλλες εντολές ή μεταβλητές) ώστε να υλοποιούνται τα παρακάτω:

α. Να εμφανίζει τον μεγαλύτερο από τους δύο ακεραίους.

β. Αν το i είναι μεγαλύτερο από 10, τότε το j να γίνεται 2, αλλιώς να γίνεται 1.

γ. Η παρακάτω εντολή να γραφεί χωρίς τη χρήση λογικών τελεστών:

```
if(i < j && i > 10 && j != 20)
        i = 5;.
```

```
#include <iostream>
int main()
{
        int i, j;

        std::cout << "Enter numbers: ";
        std::cin >> i >> j;
        /* Πρώτη σχέση. */
        if(i ____ j)
                ____;
        std::cout << i;
        /* Δεύτερη σχέση. */
        j = ____;
        /* Τρίτη σχέση. Χρησιμοποιήστε όσες γραμμές θέλετε για να
γράψετε τον κώδικά σας. */

        ____

        return 0;
}
```

Παράδειγμα

- Απάντηση. Για την πρώτη σχέση, στο πρώτο κενό η απάντηση είναι < και στο δεύτερο είναι i = j.

  Για τη δεύτερη σχέση, η απάντηση είναι (i > 10) + 1.

  Για την τρίτη σχέση μπορούμε να γράψουμε μία σειρά από ένθετες if:

```
if(i < j)
    if(i > 10)
        if(j != 20)
            i = 5;
```

Ο Τελεστής ?: (1)

- Ο τελεστής ?: επιτρέπει την εκτέλεση μίας από δύο ενέργειες, ανάλογα με την τιμή μίας έκφρασης και η σύνταξή του είναι:

```
exp1 ? exp2 : exp3;
```

- Σε μία εντολή με τον τελεστή ?: αν η έκφραση **exp1** είναι αληθής, τότε θα εκτελεστεί η έκφραση που ακολουθεί το ερωτηματικό ? (δηλαδή η **exp2**), αλλιώς θα εκτελεστεί η έκφραση που ακολουθεί την άνω-κάτω τελεία : (δηλαδή η **exp3**). Π.χ., ο παρακάτω κώδικας εμφανίζει δύο τιμές σε αύξουσα σειρά:

```
(i < j) ? (cout << i << ' ' << j) : (cout << j << ' ' << i);
```

- Η τιμή μίας έκφρασης με τον τελεστή ?: είναι ίση με την τιμή της έκφρασης που εκτελείται τελευταία. Η τιμή της έκφρασης μπορεί να αποθηκευτεί σε μία μεταβλητή. Π.χ., στον παρακάτω κώδικα αν το **k** ανήκει στο [5, 10], καταχωρείται στο **i**. Αλλιώς, το **i** γίνεται 0

```
i = (k >= 5 && k <= 10) ? k : 0;
```

Ο Τελεστής ?: (2)

- Συνήθως, ο τελεστής ?: χρησιμοποιείται για να υποκαταστήσει την εντολή **if-else**, όταν αυτή έχει απλή μορφή. Π.χ., η επόμενη **if-else** εντολή:

```
if(a > b)
        max = a;
else
        max = b;
```

μπορεί να αντικατασταθεί με: `max = (a > b)? a : b;`

- Γενικότερα, η έκφραση:

```
exp1 ? exp2 : exp3;
```

Είναι ισοδύναμη με:

```
if(exp1)
    exp2;
else
    exp3;
```

Ο Τελεστής ?: (3)

- Η έκφραση μετά την την άνω-κάτω τελεία : (δηλαδή η exp3) μπορεί να αντικατασταθεί από άλλη έκφραση που χρησιμοποιεί τον τελεστή ?:

- Π.χ.

```
k = exp1 ? exp2 : add1 ? add2 : add3;
```

Η παραπάνω έκφραση είναι ισοδύναμη με:

```
if(exp1)
        k = exp2;
else if(add1)
        k = add2;
else
        k = add3;
```

Η Εντολή switch (1)

- Η εντολή ελέγχου **switch** χρησιμοποιείται εναλλακτικά έναντι της **if-else-if** δομής, όταν θέλουμε να ελέγξουμε την τιμή μίας έκφρασης έναντι ενός σύνολου τιμών και να χειριστούμε την κάθε περίπτωση ξεχωριστά

- Μία συνηθισμένη σύνταξη της εντολής **switch**:

```
switch(έκφραση)
{
      case σταθερά_1:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης είναι ίση με τη σταθερά_1. */
      break;

      case σταθερά_2:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης είναι ίση με τη σταθερά_2. */
      break;
      ...
      case σταθερά_n:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης είναι ίση με τη σταθερά_n. */
      break;

      default:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης δεν είναι ίση με καμία από τις προηγούμενες
      σταθερές. */
      break;
}
```

Η Εντολή switch (2)

- Η έκφραση που ελέγχεται πρέπει να είναι ακέραιη μεταβλητή ή έκφραση
- Οι τιμές των σταθερά_1, σταθερά_2, ... , σταθερά_n πρέπει και αυτές να είναι ακέραιες σταθερές με διαφορετικές τιμές μεταξύ των
- Τα «βήματα» κατά την εκτέλεση της εντολής switch:
  1. Η τιμή της έκφρασης συγκρίνεται διαδοχικά με κάθε μία από τις σταθερά_1, σταθερά_2, ..., σταθερά_n
     - Αν βρεθεί μία ίδια τιμή, τότε εκτελούνται οι εντολές που ακολουθούν το αντίστοιχο case και στη συνέχεια γίνεται τερματισμός της εντολής switch μέσω της εντολής break (λεπτομέρειες για την εντολή break σε επόμενη διάλεξη...)
     - Αν δεν βρεθεί ίδια τιμή, τότε εκτελούνται οι εντολές που ακολουθούν το default και στη συνέχεια γίνεται τερματισμός της εντολής switch μέσω της εντολής break
  2. Και στις δύο περιπτώσεις, η εκτέλεση του κώδικα συνεχίζει με την πρώτη εντολή που υπάρχει μετά το άγκιστρο κλεισίματος της switch εντολής

Παράδειγμα

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
	int a;

	cout << "Enter number: ";
	cin >> a;

	switch(a)
	{
		case 1:
			cout << "One\n";
		break;

		case 2:
			cout << "Two\n";
		break;

		default:
			cout << "Other\n";
		break;
	}
	cout << "End\n";
	return 0;
}
```

Παρατηρήσεις (1)

- Η ύπαρξη της **default** περίπτωσης είναι **προαιρετική**
- Η **default** περίπτωση μπορεί να βρίσκεται οπουδήποτε μέσα σε μία εντολή **switch** (π.χ. να είναι πρώτη ή να βρίσκεται ανάμεσα στα **case**), αν όμως υπάρχει, προτείνεται να βρίσκεται στο τέλος, δηλ. μετά από κάθε **case**, ώστε να ξεχωρίζει
- Σε περίπτωση που δεν υπάρχει η **default** περίπτωση και η τιμή της έκφρασης δεν είναι ίση με κάποια από τις τιμές των σταθερά_1, σταθερά_2, ..., σταθερά_n, τότε γίνεται τερματισμός της εντολής **switch**, χωρίς να γίνει κάποια άλλη ενέργεια
  - Δηλαδή, η ροή του προγράμματος συνεχίζει με την εκτέλεση της πρώτης εντολής μετά το **switch**

Παρατηρήσεις (2)

- Η ύπαρξη της **break** σε κάθε **case** δεν είναι υποχρεωτική. Αν η **break** λείπει από την **case** που ταιριάζει, το πρόγραμμα συνεχίζει με την εκτέλεση των εντολών που υπάρχουν στις επόμενες **case**
- Συνήθως, η αθέλητη απουσία της **break** αποτελεί την πιο συνηθισμένη αιτία για την μη επιθυμητή συμπεριφορά της **switch**. Π.χ., στο προηγούμενο πρόγραμμα, αν ο προγραμματιστής δεν έχει προσθέσει την εντολή **break** στην περίπτωση του αριθμού **1** και ο χρήστης εισάγει το **1**, τότε το πρόγραμμα θα εμφανίσει **One** και **Two**
- Συνήθως, η έλλειψη της **break** αποτελεί πιθανό bug. Όταν την παραλείπετε σκόπιμα, να προσθέτετε κάποιο σχετικό σχόλιο, ώστε αν κάποιος διαβάσει τον κώδικά σας ή ακόμα και εσείς μετά από κάποιο χρονικό διάστημα να μην αναρωτιέται αν η παράλειψή της ήταν σκόπιμη ή όχι

Παρατηρήσεις (3)

- Αν θέλουμε να εκτελεστεί η ίδια ομάδα εντολών σε περισσότερες από μία **case** περιπτώσεις, μπορούμε να τις ενώσουμε

- Π.χ. αν τα μπλοκ εντολών για τις περιπτώσεις των σταθερά_1, σταθερά_2 και σταθερά_3 είναι κοινά, τότε τα αντίστοιχα **case** συνενώνονται ως εξής (έχουν, όπως βλέπουμε, κοινή **break**)

```
case σταθερά_1:
case σταθερά_2:
case σταθερά_3:
/* μπλοκ εντολών που θα εκτελεστεί αν η τιμή της έκφρασης
είναι ίση με σταθερά_1 ή σταθερά_2 ή σταθερά_3. */
break;
```

Παρατηρήσεις (4)

- Κάθε **switch** εντολή μπορεί να γραφτεί ισοδύναμα με χρήση πολλαπλών εντολών **if-else-if**
  - Σε περιπτώσεις, όμως, πολλαπλών **if-else-if** εντολών, η χρήση της **switch** μπορεί να παράξει πιο ευανάγνωστο κώδικα

- **Μειονεκτήματα** της **switch** έναντι της **if**:
  1. Η εντολή **switch** διαφέρει από την εντολή **if** στο ότι η **switch** κάνει έλεγχο μόνο για ισότητα (δηλαδή, για τιμές της έκφρασης που να είναι **ίσες** με σταθερές **case**), ενώ με την εντολή **if** μπορούμε να κάνουμε οποιαδήποτε σύγκριση
  2. Οι τιμές της **έκφρασης** της **switch** και των συγκρινόμενων σταθερών πρέπει υποχρεωτικά να είναι ακέραιες

Παράδειγμα

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
        char sign;
        int i, j;

        cout << "Enter math sign and two integers: ";
        cin >> sign >> i >> j;
        switch(sign)
        {
                case '+':
                        cout << "Sum: " << i+j << '\n';
                break;

                case '-':
                        cout << "Diff: " << i-j << '\n';
                break;

                case '*':
                        cout << "Product: " << i*j << '\n';
                break;

                case '/':
                        if(j != 0)
                                cout << "Div: " << (float)i/j << '\n';
                        else
                                cout << "Second number should not be 0\n";
                break;

                default:
                        cout << "Unacceptable operation\n";
                break;
        }
        return 0;
}
```