κεφάλαιο

4

bool, char και Άλλοι Παρόμοιοι Τύποι

Ο στόχος μας σε αυτό το κεφάλαιο:
Να κατανοήσεις τις ιδιαιτερότητες των «ειδικών» ακέραιων τύπων `bool` και `char`. Μαζί με αυτούς θα δούμε και τους παρόμοιους *απαριθμητούς* τύπους.

Προσδοκώμενα αποτελέσματα:
Θα είσαι έτοιμος για αυτά που θα μάθεις στα επόμενα δύο κεφάλαια που στηρίζονται στον τύπο `bool`. Θα μπορείς να χειριστείς στα προγράμματά σου −εκτός από αριθμούς− και χαρακτήρες (και −αργότερα− κείμενα).

Έννοιες κλειδιά:

- *τύπος* `bool`
- *συνθήκες*
- *assert*
- *λογικές παραστάσεις*
- *τύποι* `char`, `signed char`, `unsigned char`
- *απαριθμητοί τύποι*
- *μετονομασία τύπου*
- `typedef`

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Οι «Άλλοι» Ακέραιοι Τύποι:
Μέχρι τώρα γράφουμε τις συνθήκες επαλήθευσης σε σχόλια, για να παρακολουθούμε τη λογική του προγράμματός μας. Τώρα θα δούμε ότι μπορούμε:

- να τυπώνουμε την τιμή (1 για `true` ή 0 για `false`) μιας συνθήκης, με μια `"cout << …"`,
- να φυλάγουμε την τιμή μιας συνθήκης σε μια μεταβλητή,

και (στο επόμενο κεφάλαιο):

- να ρυθμίζουμε την εκτέλεση του προγράμματός μας με βάση τις τιμές συνθηκών.

Αυτά έχουν να κάνουν με τον τύπο `bool`.
Θα δούμε ακόμη τους τύπους `char` και `wchar_t`.

Στο Κεφ. 2 είπαμε ότι όλοι αυτοί είναι ακέραιοι τύποι. Τι θα πει αυτό; Ας πούμε ότι έχουμε:

```
bool b( true );
char c1( ' ' ), c2( '!' ), c3;
int j;
```

και δίνουμε:

```
j = 32*b;
c3 = c1 + c2;
cout << j << "  " << c3 << endl;
```

Αποτέλεσμα:

```
32  A
```

Στη συνέχεια θα καταλάβεις πώς βγαίνουν αυτά τα αποτελέσματα αλλά και γιατί δεν πρέπει να γράφεις τέτοιες εντολές!

Πριν αρχίσεις τη μελέτη της §4.1 ρίξε μια ματιά στο Παρ. Α και μη διστάζεις να το συμβουλεύεσαι όσο θα μελετάς τις τρεις πρώτες παραγράφους.

4.1 Οι Συνθήκες στο Πρόγραμμα

Σε όσα είπαμε μέχρι τώρα είδαμε και μερικές συνθήκες, π.χ.:

$$(number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

$$(a == a_0) \;\&\&\; (b == b_0) \;\&\&\; (|a_0 + b_0| \le ½)$$

$$(|tP - \sqrt{2h/g}\,| < \varepsilon_{TR}\sqrt{2h/g}\,) \;\&\&\; (|vP + \sqrt{2gh}\,| < \varepsilon_{VR}\sqrt{2gh}\,)$$

Σε όλες τις περιπτώσεις (εκτός από την πρώτη) οι συνθήκες που είδαμε είναι συγκρίσεις που συνδέονται μεταξύ τους με λογικές πράξεις. Όποτε βάλαμε τις συνθήκες μέσα στο πρόγραμμά μας ήταν σε σχόλια. Η C++ μπορεί να «κατανοήσει» και να χρησιμοποιήσει μια συνθήκη αν γραφεί σύμφωνα με τους κανόνες που βάζει η γλώσσα:

- Στις συγκρίσεις, αντί των συμβόλων που χρησιμοποιούν τα μαθηματικά, χρησιμοποίησε αυτά που θέλει η C++. Δες το Πλ. 4.1.
- Στις λογικές πράξεις γράψε τους τελεστές όπως δίνονται στο Πλ. 4.2 αλλά πρόσεξε μερικές «επιφυλάξεις» που παραθέτουμε στη συνέχεια. Μπορείς να χρησιμοποιείς τα "`&&`", "`||`", "`!`", αντί των "$\wedge$", "$\vee$", "$\neg$" αντιστοίχως, αλλά αντί του "$\Leftrightarrow$" θα χρησιμοποιείς το "`==`", αντί του "$\Rightarrow$" το "`<=`" και αντί του "`xor`" το "`!=`". Ο Πίν. 4-1 είναι στην πραγ-

Τιμές Ορισμάτων		Αποτελέσματα Πράξεων					
P	Q	!P (not P)	P && Q (P and Q)	P || Q (P or Q)	P <= Q (P ⇒ Q)	P == Q (P ⇔ Q)	P != Q (P xor Q)
false	false	true	false	false	true	true	false
false	true	true	false	true	true	false	true
true	false	false	false	true	false	false	true
true	true	false	true	true	true	true	false

Πίν. 4-1 Οι πίνακες αλήθειας όλων των λογικών πράξεων που μας δίνει η C++.

Πλαίσιο 4.1

Τελεστές Συσχέτισης (Σύγκρισης)

Μαθηματικά	C++	Όνομα και σημασία του τελεστή συσχέτισης
=	`==`	ίσο με (π.χ. `x == 5`)
≠	`!=`	διάφορο
<	`<`	μικρότερο (π.χ. `x < 12`)
>	`>`	μεγαλύτερο
≤	`<=`	μικρότερο ή ίσο
≥	`>=`	μεγαλύτερο ή ίσο

ματικότητα σύντμηση των πινάκων του Παρ. Α.

- Αντικατάστησε τις διπλές (ή πολλαπλές) συγκρίσεις με απλές,

  π.χ. *μην γράφεις* *INT_MIN ≤ number ≤ INT_MAX*

  αλλά `INT_MIN <= number && number <= INT_MAX`

Η C++ σου επιτρέπει αντί των "`&&`", "`||`", "`!`" να χρησιμοποιείς τα "`and`", "`or`", "`not`" αντιστοίχως. Εμείς θα χρησιμοποιούμε εδώ τα "`&&`", "`||`", "`!`" μια και είναι αυτά που χρησιμοποιούνται συνήθως (από τον καιρό της C) και αυτά θα συναντάς όποτε διαβάζεις άλλα βιβλία. Άλλωστε, είναι πολύ πιθανό, η C++ που χρησιμοποιείς (π.χ. η Borland C++) να μη δέχεται τα "`not_eq`" (αντί για το "`!=`"), "`and`", "`or`", "`not`".

Τα "`==`", "`<`", "`>`", "`<=`", "`>=`", "`!=`", "`not_eq`", "`&&`", "`and`", "`||`", "`or`", "`!`", "`not`" είναι λεξικά σύμβολα (tokens) της C++ και δεν θα πρέπει να διαχωρίζεις τους χαρακτήρες τους, με οποιονδήποτε τρόπο, όταν τα γράφεις ενώ από την άλλη θα πρέπει να τα διαχωρίζεις από τα προηγούμενα και τα επόμενά τους.

Οι **λογικές παραστάσεις** (logical expressions), που η τιμή τους είναι "`true`" ή "`false`", κατασκευάζονται με κανόνες παρόμοιους με αυτούς που είδαμε για τις αριθμητικές παραστάσεις. Μπορούμε να πούμε, ότι μιά λογική παράσταση είναι ένας συνδυασμός από λογικές σταθερές ("`true`" ή "`false`") και από συσχετίσεις (π.χ. συγκρίσεις αριθμητικών τιμών), που συνδέονται μεταξύ τους (αν υπάρχουν περισσότερα ορίσματα) με τους λογικούς τελεστές.

Στο Πλ.4.2 βλέπεις ένα συντακτικό κανόνα *για τις παρενθέσεις*. Μας λέει ότι μπορούμε να απλουστεύουμε τη γραφή λογικών παραστάσεων παραλείποντας παρενθέσεις.

Δηλαδή: *Μπορείς να γράψεις*

```
!(a > 0) && b <= 0 || c == 1
```

που είναι το ίδιο με το

```
((!(a > 0)) && (b <= 0)) || (c == 1)
```

Πάντως η δεύτερη μορφή είναι προτιμότερη.

Αν όμως θέλεις να γράψεις

```
(!(a > 0) && b <= 0) xor (c == 1)
```

θα *πρέπει να κρατήσεις τις παρενθέσεις*. Διότι αν γράψεις:

```
!(a > 0) && b <= 0 != c == 1
```

θα υπολογιστεί η `!(a > 0) && (((b <= 0) != c) == 1)` (δες τον Πίν. 4-2).

Ο ίδιος κανόνας υπάρχει, με άλλον τρόπο, και στον πίνακα του Παρ. Ε, και μας λέει ότι:

- πρώτα υπολογίζονται οι αρνήσεις "`!`" ("`not`") (προτ. 3),
- υπολογίζονται οι αριθμητικές παραστάσεις,
- στη συνέχεια γίνονται οι συγκρίσεις, πρώτα οι "`<`", "`<=`", "`>`", "`>=`" (προτ. 5) και μετά οι "`==`", "`!=`" (προτ. 6),

Πλαίσιο 4.2

Λογικοί Τελεστές

Λογική	C++
∧, **and**, **και**	`&&` `and`
∨, **or**, **ή**	`||` `or`
¬, ~, **όχι**	`!` `not`
⊻, **xor**	`!=`
⇒, ⊃	`<=`
⇔	`==`

Συντακτικός κανόνας:

- ***Στις λογικές πράξεις επιτρέπεται να μη βάλεις παρενθέσεις, αλλά να ξέρεις ότι οι πράξεις γίνονται με την εξής σειρά: πρώτα οι not (!), μετά οι and (&&), και τέλος οι or (||). Για τις "!=" (xor), "<=" (⇒) και "==" (⇔) χρειάζονται παρενθέσεις.***

- μετά γίνονται οι λογικές πράξεις με τη σειρά: "`&&`" ("`and`") (προτ. 10) και "`||`" ("`or`")[1] (προτ. 11),
- τέλος γίνονται οι εκχωρήσεις.

Μερικά παραδείγματα λογικών παραστάσεων:

```
(x > 0) || (y > 0)
(x > 0) == (y > 0)
(x < 0) == (y < 0)
(k <= l) && (l < m + 5)
(a – 2 > b) && (b < c) || (c == 0)
(a – 2 > b) && ((b < c) || (c == 0))
```

Βέβαια η τιμή όλων αυτών των παραστάσεων, `true` ή `false`, εξαρτάται από τη συγκεκριμένη τιμή των λογικών ορισμάτων. Αν π.χ. υποθέσουμε ότι τα αριθμητικά ορίσματα (τύπου `int`) έχουν τιμές: $x = 5$, $y = 4$, $a = -1$, $b = c = 0$, $k = -2$, $l = 1$ και $m = -4$, τότε οι προηγούμενες λογικές παραστάσεις θα πάρουν τις εξής τιμές:

`(x > 0) || (y > 0)`: `true` (επειδή $x > 0$ και $y > 0$),
`(x > 0) == (y > 0)`: `true` (επειδή $(x > 0)$ `true` και $(y > 0)$ `true`),
`(x < 0) == (y < 0)`: `true` (επειδή $(x < 0)$ `false` και $(y > 0)$ `false`),
`(k <= l) && (l < m + 5)`: `false` (επειδή: $(k <= l)$ `true` ενώ $(l < m + 5)$ `false`),
`(a – 2 > b) && (b < c) || (c == 0)`: `true` (επειδή c `==` 0),
`(a – 2 > b) && ((b < c) || (c == 0))`: `false` (επειδή $(a - 2 > b)$ `false`).

Πρόσεξε τη 2η και την 3η: το `==` υλοποιεί την αμοιβαία συνεπαγωγή (⇔). Και οι δύο παραστάσεις έχουν τιμή `true` διότι και στις δύο περιπτώσεις οι ανισότητες έχουν την ίδια τιμή: στη 2η και οι δύο είναι `true`, στην 3η και οι δύο είναι `false`.

Πρόσεξε ακόμη την 5η και την 6η:

- Αφού η `&&` υπολογίζεται πριν από την `||`, η 5η ισοδυναμεί με: `((a – 2 > b) && (b < c)) || (c == 0)` και επειδή `true ||` οτιδήποτε μας κάνει `true` και η `c == 0` είναι `true`, όλη η παράσταση είναι `true`.
- Στην 6η, με τις παρενθέσεις ζητούμε να υπολογιστεί πρώτα η `||` και μετά η `&&`. Επειδή `false &&` οτιδήποτε μας κάνει `false` και η `a – 2 > b` είναι `false`, όλη η παράσταση είναι `false`.

[1] Εδώ υπάρχει ένα λεπτό σημείο που θα το δούμε στο επόμενο κεφάλαιο.

Παίρνοντας υπόψη την προτεραιότητα των πράξεων μπορούμε να βγάλουμε μερικές παρενθέσεις και τα παραδείγματά μας να γραφούν:

```
x > 0 || y > 0
x > 0 == y > 0
x < 0 == y < 0
k <= l && l < m + 5
a - 2 > b && b < c || c == 0
a - 2 > b && (b < c || c == 0)
```

Και τώρα πρόσεξε:

- ***Σε μια εντολή εξόδου (`cout << …`) μπορείς να βάζεις ως όρισμα μια συνθήκη, μέσα σε παρενθέσεις. Αυτό που θα γίνει είναι το εξής: θα υπολογισθεί η τιμή της και αν είναι "`true`" (αν ισχύει) θα τυπωθεί η τιμή "1" (ένα), ενώ αν είναι "`false`" (αν δεν ισχύει) θα τυπωθεί η τιμή "0" (μηδέν).***

Με άλλα λόγια, αν *Π* μια λογική παράσταση τότε η εντολή:

```
cout << (Π) …
```

ισοδυναμεί με:

```
cout << 1 …
```
αν η *Π* έχει τιμή `true`, και με
```
cout << 0 …
```
αν η *Π* έχει τιμή `false`.

Παράδειγμα

Αν έχουμε δηλώσει:

```
int  n1, n2;
```

τότε οι εντολές:

```
n1 = 12;  n2 = 13;
cout << n1 << " < " << n2 << " : " << (n1 < n2) << endl;
cout << n1 << " > " << n2 << " : " << (n1 > n2) << endl;
```

θα δώσουν:

```
12 < 13 : 1
12 > 13 : 0
```

Στην επόμενη παράγραφο βλέπεις πώς χρησιμοποιούμε αυτή τη δυνατότητα για να ελέγχουμε την ορθότητα ενός προγράμματος.

Προς το παρόν όμως δες κάτι που μπορεί να σου συμβεί: Είπαμε πιο πάνω ότι για να «κατανοήσει» η C++ μια λογική παράσταση πρέπει να αντικαταστήσεις «*τις διπλές (ή πολλαπλές) συγκρίσεις με απλές*», π.χ. αντί για `0 <= x < 10` γράψε `(0 <= x) && (x < 10)`. Αν δεν το κάνεις, ο μεταγλωττιστής θα δεχθεί τη διπλή σύγκριση αλλά θα την «κατανοήσει» ως `(0 <= x) < 10` που έχει πάντοτε τιμή `true` διότι το `(0 <= x)` είναι μικρότερο από 10 είτε έχει τιμή `true` (1) είτε έχει τιμή `false` (0).

4.1.1 Το Λάθος που θα Κάνεις Συχνά!

Ο τίτλος της παραγράφου είναι απαισιόδοξος, αλλά είναι παρατηρημένο: οι αρχάριοι στη C++ (και στη C) κάνουν το ίδιο *σοβαρότατο* σφάλμα: προσπαθούν (μάταια) να συγκρίνουν για ισότητα με το "`=`" αντί για το "`==`". Κοίταξε όμως τι μπορεί να συμβεί:

Ας πούμε ότι έχεις: *a* `==` –1, *b* `==` *c* `==` 0 και θέλεις να τυπώσεις την τιμή της παράστασης:

```
(a – 2 > b) && (b < c) || (c == 0)
```

και δίνεις:

```
cout << ((a – 2 > b) && (b < c) || (c = 0)) << endl;
```

Αποτέλεσμα: **0** (**false**)! Πώς έγινε αυτό; Η C++ υπολογίζει την τιμή της *εκχώρησης* [2] **c = 0** που είναι μηδέν και όταν έλθει η ώρα της λογικής πράξης **||** θεωρεί το 0 ως **false**.

Ας πούμε τώρα ότι θέλεις την τιμή της παράστασης

```
(a – 2 > b) && (b < c) || (c == 44)
```

και δίνεις:

```
cout << ((a – 2 > b) && (b < c) || (c = 44)) << endl;
```

Αποτέλεσμα: **1** (**true**)! Εδώ τι έγινε; Η C++ υπολογίζει την τιμή της *εκχώρησης* **c = 44** που είναι 44 και όταν έλθει η ώρα της λογικής πράξης **||** θεωρεί το 44 ≠ 0 ως **true**. Εκτός από αυτό όμως, η *c*, για το υπόλοιπο πρόγραμμά σου, έχει τιμή 44 αντί για 0 που θα ήθελες να έχει!

Και στις δύο περίπτώσεις η Borland C++ θα σου δώσει *προειδοποίηση*: «**Possibly incorrect assignment**» (πιθανόν εσφαλμένη εκχώρηση). Καλό είναι λοιπόν *να προσέχεις*, όχι μόνον τα *λάθη*, *αλλά* και *τις προειδοποιήσεις*.

4.2 Οι Τιμές των Συνθηκών Επαλήθευσης

Αν ένα πρόγραμμά μας είναι σωστό τότε, κάθε φορά που εκτελείται, όλες οι συνθήκες επαλήθευσης θα (πρέπει να) έχουν τιμή **true**. Αν αυτό δεν συμβαίνει τότε κάτι δεν πάει καλά.

Όπως καταλαβαίνεις, σύμφωνα με όσα είπαμε στην προηγούμενη παράγραφο μπορούμε να ζητήσουμε από τον υπολογιστή να υπολογίσει και να μας τυπώσει τις τιμές των συνθηκών επαλήθευσης.

Δες ένα παράδειγμα με το πρόγραμμα της αντιμετάθεσης:

```
#include <iostream>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;
   int S;

   a = a0;  b = b0;
// a == a0 && b == b0
   cout << " Προ: " << (a == a0 && b == b0) << endl;
   S = a;
// (b == b0) && (S == a0)
   cout << "   1: " << ((b == b0) && (S == a0)) << endl;
   a = b;
// (a == b0) && (S == a0)
   cout << "   2: " << ((a == b0) && (S == a0)) << endl;
   b = S;
// a == b0 && b == a0
   cout << " Απτ: " << (a == b0 && b == a0) << endl;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Πρόσεξε τα εξής:

- Κάτω από κάθε σχόλιο με συνθήκη επαλήθευσης υπάρχει μια "**cout << …**" που τυπώνει την τιμή της συνθήκης. Οι εκτυπώσεις αρχίζουν με ένα χαρακτηριστικό: « **Προ:**» για την προϋπόθεση, « **Απτ:**» για την απαίτηση και αριθμούς 1, 2, 3,... για τις ενδιάμεσες συνθήκες.

[2] «τιμή της *εκχώρησης*»!; Ναι, υπάρχει τέτοιο πράγμα και θα το δούμε αργότερα· προς το παρόν: είναι η τιμή που αποθηκεύεται στη μεταβλητή.

- Πρόσεξε πώς έγινε η μετάφραση των συνθηκών σε C++, σύμφωνα με τους κανόνες που είπαμε.

  Να ένα παράδειγμα εκτέλεσης:

```
Προ: 1
  1: 1
  2: 1
Απτ: 1
a = 20   b = 10
```

  Ας δούμε τώρα το πρώτο, εσφαλμένο, πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;

   a = a0;  b = b0;
// a == a0 && b == b0
   cout << " Προ: " << (a == a0 && b == b0) << endl;
   a = b;  b = a;
// a == b0 && b == a0
   cout << " Απτ: " << (a == b0 && b == a0) << endl;
   cout << " a = " << a << "   b = " << b << endl;
}
```

Παράδειγμα εκτέλεσης:

```
Προ: 1
Απτ: 0
a = 20   b = 20
```

Προσοχή! Να υπενθυμίσουμε ότι με τύπους κινητής υποδιαστολής μπορεί να έχεις και μερικές εκπλήξεις όταν οι συνθήκες σου έχουν ισότητες[3].

Βέβαια, θα πρέπει να (ξανα)τονίσουμε ότι το να πάρεις μερικά παραδείγματα εκτέλεσης με όλες τις συνθήκες `true` δεν σημαίνει ότι το πρόγραμμά σου είναι σωστό.

4.3 Έλεγχος Συνθηκών Επαλήθευσης: *assert()*

Στο επόμενο κεφάλαιο θα μάθουμε πώς να ελέγχουμε τη ροή της εκτέλεσης του προγράμματος. Προς το παρόν όμως θα δούμε έναν τρόπο για να αποφασίζουμε αν θα *διακόψουμε* την εκτέλεση του προγράμματος σε σχέση με τις τιμές των συνθηκών επαλήθευσης.

Δες έναν άλλον τρόπο για να γράψουμε το δεύτερο πρόγραμμα της προηγούμενης παραγράφου:

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;

   a = a0;  b = b0;
   assert( a == a0 && b == b0 );
   a = b;  b = a;
   assert( a == b0 && b == a0 );
   cout << " a = " << a << "   b = " << b << endl;
}
```

[3] Για δοκίμασε να κάνεις τα ίδια με το πρόγραμμα της ελεύθερης πτώσης. Χα!

Η εκτέλεση του προγράμματος δίνει:

```
Assertion failed: a == b0 && b == a0, file test02a.cpp, line 13
```

Τι είναι αυτή η «παράξενη» εντολή "`assert(`*συνθήκη*`)`"; Να τη σκέφτεσαι ως κλήση συνάρτησης, όπως αυτές που μάθαμε στο Κεφ. 1, που όμως:

- παίρνει όρισμα μια *συνθήκη* (ή μια τιμή τύπου `bool`, όπως θα μάθουμε στην επόμενη παράγραφο) και
- *δεν επιστρέφει* κάποια *τιμή*.

Για τέτοιες συναρτήσεις θα μιλήσουμε αργότερα.

Για να χρησιμοποιήσεις την *assert* θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <cassert>`".

Ας δούμε τη λειτουργία της:

- Την πρώτη φορά που καλείται, στη γραμμή 11, η συνθήκη-όρισμα ισχύει (`true`). Ως προς τα αποτελέσματα: είναι σαν να μην υπάρχει.
- Τη δεύτερη φορά καλείται, στη γραμμή 13, η συνθήκη-όρισμα δεν ισχύει (`false`). Ως αποτέλεσμα διακόπτεται η εκτέλεση του προγράμματος και παίρνουμε αυτά που είδες.

Ξαναγράφουμε και το πρώτο πρόγραμμα ως εξής:

```
#include <iostream>
#include <cassert>
using namespace std;
int main()
{
   const int a0 = 10,  b0 = 20;

   int a, b;
   int S;

   a = a0;  b = b0;
   assert( a == a0 && b == b0 );
   S = a;
   assert( (b == b0) && (S == a0) );
   a = b;
   assert( (a == b0) && (S == a0) );
   b = S;
   assert( a == b0 && b == a0 );
   cout << " a = " << a << "   b = " << b << endl;
}
```

Αποτέλεσμα:

```
 a = 20   b = 10
```

Καμιά ένδειξη για την ύπαρξη της *assert*![4]

Τέλος, ας δούμε πώς μπορούμε να χρησιμοποιήσουμε την *assert* για τη διασφάλιση της προϋπόθεσης του προγράμματος της ελεύθερης πτώσης (§2.7). Στο αρχείο `eleptw.cpp` έχουμε το πρόγραμμά μας που τώρα, με την εισαγωγή της "`assert( h >= 0 )`" στη 14η γραμμή, έχει την εξής μορφή:

```
#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης
```

[4] Παρ' όλα αυτά η εκτέλεση του προγράμματος γίνεται πιο αργή. Θα δούμε αργότερα ότι αυτό διορθώνεται.

```
// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;

   assert( h >= 0 );

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
// Υπολόγισε τα tP, vP
   tP = sqrt( 2*h/g );
   vP = -sqrt(2*h*g);
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
// Τύπωσε τα tP, vP
   cout << " Αρχικό ύψος = " << h << " m" << endl;
   cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
   cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
        << vP << " m/sec" << endl;
}
```

Δες ένα παράδειγμα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος σε m: -80
Assertion failed: h >= 0, file eleptw.cpp, line 14
```

Φυσικά, αν δίναμε αρχικό ύψος **80** το αποτέλεσμα θα ήταν ακριβώς αυτό που ξέραμε από το αρχικό πρόγραμμα.

4.4 Ο Τύπος bool

Ας δούμε τώρα πώς μπορούμε «*να φυλάγουμε την τιμή μιας συνθήκης σε μια μεταβλητή*», όπως λέγαμε στην §4.1.

Στα προηγούμενα κεφάλαια γνωρίσαμε τον τρόπο γραφής και δήλωσης δύο τύπων στοιχείων: **int** (ακέραιοι) και **double** (πραγματικοί). Στην παράγραφο αυτή θα γνωρίσουμε έναν τρίτο τύπο στοιχείων, που λέγεται τύπος **bool**. Οι τιμές του τύπου **bool** είναι οι **false** και **true** και είναι διαταγμένες:

```
false < true
```

Μεταξύ τιμών τύπου **bool** μπορεί να γίνονται οι λογικές πράξεις που μάθαμε: η "**&&**", η "**||**", η "**!**". Μα είπαμε ότι μπορεί να γίνονται και οι άλλες: ⇒, ⇔, **xor**! Ναι, αλλά να καταλάβουμε τι συμβαίνει: η υλοποίησή τους στηρίζεται στη διάταξη που δώσαμε παραπάνω. Πήγαινε στο Παρ. Α και

- Δες τον αληθοπίνακα της "⇒" (Πίν. Α-3)· στις γραμμές 1, 2 και 4 για τις οποίες η $P \Rightarrow Q$ έχει τιμή **true**, με βάση τη διάταξη **false** < **true**, ισχύει η P **<=** Q. Στη γραμμή 3, όπου η $P \Rightarrow Q$ έχει τιμή **false**, έχουμε $P > Q$ ή αλλιώς **!**(P **<=** Q).
- Δες τον αληθοπίνακα της "**xor**" (Πίν. Α-4)· στις γραμμές 2 και 3 για τις οποίες η P **xor** Q έχει τιμή **true**, έχουμε P **!=** Q, ενώ στις 1 και 4 όπου η P **xor** Q έχει τιμή **false**, P **==** Q ή **!**(P **!=** Q).
- Τέλος, δες τον αληθοπίνακα της "⇔" (Πίν. Α-5)· στις γραμμές 1 και 4 για τις οποίες η $P \Leftrightarrow Q$ έχει τιμή **true**, έχουμε P **==** Q, ενώ στις 2 και 3 όπου η $P \Leftrightarrow Q$ έχει τιμή **false**, P **!=** Q ή **!**(P **==** Q).

Μπορούμε λοιπόν να συμπληρώσουμε αυτό που είπαμε πιο πάνω: μιά λογική παράσταση είναι ένας συνδυασμός από λογικές σταθερές (**true** ή **false**), μεταβλητές τύπου **bool** και από συσχετίσεις (ή συγκρίσεις αριθμητικών τιμών), που συνδέονται μεταξύ τους (αν υπάρχουν περισσότερα ορίσματα) με τους τελεστές "**&&**", "**||**", "**!**", "**<=**", "**==**" και "**!=**". Ακόμη, σε μια μεταβλητή τύπου **bool** μπορείς να αποθηκεύεις την τιμή μιας συνθήκης (λογικής παράστασης).

Οι μεταβλητές τύπου **bool** δηλώνονται στο πρόγραμμα, όπως και οι μεταβλητές των άλλων τύπων που γνωρίσαμε, στο μέρος δήλωσης μεταβλητών. Π.χ.:

```
   bool   x, y, logical, signal, ok, lgc;
   int    i, k;
```

```
   double a;
```

Η παραπάνω δήλωση σημειώνει ότι κατά την εκτέλεση του προγράμματος οι μεταβλητές *x*, *y*, *logical*, *signal*, *ok* και *lgc* θα παίρνουν τις τιμές **true** ή **false**. Έτσι π.χ., μπορούμε να γράψουμε μέσα στο πρόγραμμα τις παρακάτω εντολές εκχώρησης, όπου η δεξιά πλευρά είναι μια λογική παράσταση (ή μια λογική σταθερά), ενώ η αριστερή πλευρά είναι μια μεταβλητή τύπου **bool**.

```
    x = true;                       y = !signal || x;
    logical = (x || y) && signal;   ok = (a <= 18) && (k != 2);
    lgc = false;                    signal = k < -1;
```

Παράδειγμα

Το παρακάτω πρόγραμμα τυπώνει το περιεχόμενο του Πίν. 4-1, μόνο που αντί για **false** και **true** έχει 0 και 1 αντίστοιχα.

```
#include <iostream>
using namespace std;
int main()
{
   bool  P, Q;

   cout << "P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q" << endl;
   P = false;  Q = false;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = false;  Q = true;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = true;   Q = false;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
   P = true;   Q = true;
   cout << P << "   " << Q << "   " << !P << "   " << (P && Q)
        << "    " << (P || Q) << "    " << (P <= Q) << "    "
        << (P == Q) << "    " << (P != Q) << endl;
}
```

Αποτέλεσμα:

```
P   Q  !P P&&Q P||Q P<=Q P==Q P!=Q
0   0   1   0    0    1    1    0
0   1   1   0    1    1    0    1
1   0   0   0    1    0    0    1
1   1   0   1    1    1    1    0
```

Αφού όταν ζητάμε να τυπωθεί **true** ή **false** παίρνουμε 1 ή 0 αντιστοίχως καταλαβαίνεις και γιατί (στο πρόγραμμα της εισαγωγής) το **32*true** μας δίνει 32! Ο **bool** είναι ένας ακέραιος τύπος με δύο τιμές 0 και 1. Αυτό θα *πρέπει να προσπαθείς να το ξεχάσεις* αν και θα υπάρχουν πολλά για σου το θυμίζουν.

- Να θυμάσαι μόνον ότι **false < true** (και όχι τα 0 και 1).
- Να θυμάσαι ότι **!false == true** και **!true == false** (και όχι ότι **1-false == true** και **1-true == false**).

Να δούμε τώρα τις κληρονομιές από τη C: Στη C δεν υπάρχει τύπος **bool** και *οποιαδήποτε τιμή δεν είναι μηδέν θεωρείται ως "***true***", ενώ το "***0***" θεωρείται ως "***false***"*. Η C++ είναι συμβατή με αυτά. Αν έχεις δηλώσει:

```
   bool b;
```

τότε οι εντολές:

```
   b = static_cast<bool>( 37.5 );  cout << b << "  ";
```

```
   b = 37.5;                        cout << b << "  ";
   b = static_cast<bool>( 0 );      cout << b << "  ";
   b = 0;                           cout << b << endl;
```

δίνουν:

```
1  1  0  0
```

Δηλαδή, η μη μηδενική τιμή (τύπου **double**) γίνεται **true** με ανοικτή ή εννοούμενη τυποθεώρηση ενώ το 0 γίνεται **false**.

Καλώς ή κακώς, εργαλεία από τη C θα χρησιμοποιούμε συχνά και δεν γίνεται να ξεφύγουμε από χρήσεις μη λογικών τιμών σε θέσεις όπου περιμένουμε τιμή τύπου **bool**. Εμείς πάντως θα αποφεύγουμε παρόμοιες χρήσεις. Όσο είναι δυνατό.

Σημείωση: ▶

Για τις μεταβλητές τύπου **bool** θα δεις πολύ συχνά τον όρο **σημαία** (flag) που μπορεί να είναι *ανεβασμένη* (**true**) ή *κατεβασμένη* (**false**). ◀

4.4.1 Για να Γράφουμε "false" και "true"

Αν τα "**0**" και "**1**" σε ενοχλούν πολύ και θέλεις να γράφεις "**false**" και "**true**" μπορείς να στείλεις προς το *cout* το μήνυμα "**boolalpha**" και όλα θα αλλάξουν με μιας: Το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
  cout << false << "  " << true << endl;
  cout << boolalpha;
  cout << false << "  " << true << endl;
}
```

θα δώσει:

```
0  1
false  true
```

4.5 Οι Τύποι char

Στην §2.3 μεταξύ των άλλων ακέραιων τύπων είδαμε και τον τύπο **char**. Στην πραγματικότητα η C++ έχει τρεις διαφορετικούς τύπους:

- **char**,
- **signed char**,
- **unsigned char**.

Ας δούμε ένα μικρό αλλά χαρακτηριστικό

Παράδειγμα 1 ➴

```
#include <iostream>
using namespace std;
int main()
{
   char c;     unsigned char u;     signed char s;

   c = 195;  u = 195;  s = 195;
   cout << c << "  " << u << "  " << s << endl;
   cout << static_cast<int>(c) << "  " << static_cast<int>(u)
        << "  " << static_cast<int>(s) << endl;
}
```

Αποτέλεσμα:

```
Γ  Γ  Γ
-61  195  -61
```

Ας τα πάρουμε από την αρχή:

- Με τις εντολές: `c = 195;  u = 195;  s = 195` δίνουμε και στις τρεις μεταβλητές την ίδια ακέραιη τιμή: 195. Να τονίσουμε βέβαια ότι αυτές οι εντολές εκτελούνται ως:

```
c = static_cast<char>(195);
u = static_cast<unsigned char>(195);
s = static_cast<signed char>(195) .
```

- Το αποτέλεσμα της `cout << c << "  " << u << "  " << s << endl` είναι λίγο αναπάντεχο: `Γ  Γ  Γ`. Πώς εξηγείται; Στο Παρ. Δ, Πίν. Δ-4 (Δ-3) μπορείς να δεις ότι στο σύνολο χαρακτήρων για τα Windows, στη θέση 195, υπάρχει το ελληνικό γράμμα Γ.
- Η δεύτερη γραμμή είναι εντελώς ανεξήγητη για όποιον δεν ξέρει αριθμητική modulo (mod 256 στην περίπτωσή μας). Απλώς παρατήρησε ότι 61 = 256 – 195.

&

Τώρα, ξεχνούμε για λίγο το ότι βάλαμε τον `char` στους ακέραιους τύπους και ξεκινούμε από το εξής: *στον τύπο* `char` *παριστάνουμε χαρακτήρες*· όλους τους χαρακτήρες που περιλαμβάνει το σύνολο χαρακτήρων του υπολογιστή.

Παράδειγμα 2

Μέσα στο πρόγραμμά μας μπορεί να δηλώσουμε:

```
const char plus = '+', space = ' ', aLower = 'a', aUpper = 'A';
char letter, digit, spcChar;
```

και στη συνέχεια να δώσουμε:

```
letter = aLower;
digit = '7';
spcChar = plus;
cout << ' ' << letter << space << digit << spcChar << endl;
```

Αποτέλεσμα:

```
 a 7+
```

ή, για να το δεις καλύτερα:

```
␣a␣7+
```

&

Οι σταθερές τύπου `char` έχουν τη μορφή που βλέπεις στα παραδείγματα: ένας χαρακτήρας μεταξύ δύο αποστρόφων:

```
'+'   ' '   'a'   'A'   '7'
```

Όπως είναι φανερό, έχεις πρόβλημα να παραστήσεις την απόστροφο· η λύση είναι αυτή που λέγαμε και για τους ορμαθούς χαρακτήρων: αν θέλεις να γράψεις σταθερές τύπου `char` που έχουν ως τιμή κάποιον από τους χαρακτήρες: `\`, `?`, `'`, `"` θα πρέπει να γράψεις: `'\\'`, `'\?'`, `'\''`, `'\"'`. Με παρόμοιο τρόπο (Πίν. 4-2) μπορείς να παραστήσεις και *μη εκτυπώσιμους* χαρακτήρες.

Στο παράδειγμά μας είδαμε ότι σε μια μεταβλητή τύπου `char` μπορούμε να δίνουμε τιμές με εντολές εκχώρησης. Μπορούμε να διαβάζουμε και από το πληκτρολόγιο; Βεβαίως.

LF	`'\n'`
HT	`'\t'`
VT	`'\v'`
BS	`'\b'`
CR	`'\r'`
FF	`'\f'`
BEL	`'\a'`
\	`'\\'`
?	`'\?'`
'	`'\''`
"	`'\"'`

Πίν. 4-2 Πώς γράφουμε ως σταθερές τύπου `char` μερικούς χαρακτήρες εκτυπώσιμους και μη. Για τους μη εκτυπώσιμους δες το Παρ. D.

Παράδειγμα 3

Ας πούμε ότι έχουμε:

```
char c1, c2, c3;
   :
cin >> c1 >> c2 >> c3;
cout << c1 << c2 << c3 << endl;
```

Πληκτρολογούμε:

a b c<Enter>

και παίρνουμε:

```
abc
```

Θαυμάσια! Αλλά… Δεν είπαμε ότι το κενό (διάστημα) είναι χαρακτήρας; Γιατί δεν έγινε τιμή της *c2*; Γι' αυτό φταίει ο "`>>`" του *cin* που «τρώει» τα διαστήματα, τις αλλαγές γραμμής και άλλα παρόμοια. Αν θέλεις να διαβάζεις τα πάντα χρησιμοποίησε τη

```
cin.get(a);
```

που διαβάζει από το πληκτρολόγιο έναν χαρακτήρα και τον αποθηκεύει ως τιμή της *a*, τύπου `char`. Αλλά εδώ προσοχή: *πρέπει να διαβάζεις τα πάντα*!

Παράδειγμα 3

Ας πούμε ότι έχουμε:

```
char c1, c2, c3, c4;

cin.get(c1);  cin.get(c2);
cin.get(c3);  cin.get(c4);
cout << " c1: " << c1 << endl;
cout << " c2: " << c2 << endl;
cout << " c3: " << c3 << endl;
cout << static_cast<int>(c4) << endl;
```

Πληκτρολογούμε:

```
a b<enter>
```

και παίρνουμε:

```
 c1: a
 c2:
 c3: b
10
```

Στη δεύτερη γραμμή έχουμε στην πραγματικότητα: "`c2: ␣`". Να το διάστημα που λέγαμε. Το 10 στην τέταρτη γραμμή είναι αποτέλεσμα της: `cout << static_cast<int>(c4)`. Στη *c4* αποθηκεύτηκε το <Enter> (LF, '`\n`') που δώσαμε στο τέλος της γραμμής!

Από το τελευταίο παράδειγμα θα πρέπει να καταλαβαίνεις ότι αν στο ίδιο πρόγραμμα ανακατέψεις "`cin >> …`" και "`cin.get(v)`" θα έχεις προβλήματα.

- ***Σε ένα πρόγραμμα θα δουλεύεις είτε με "`cin >> …`" είτε με "`cin.get(v)`". Στη δεύτερη περίπτωση θα παίρνεις ό,τι στέλνει το πληκτρολόγιο προς το πρόγραμμά σου. Για να τις ανακατέψεις θα πρέπει να ξέρεις πώς ακριβώς δουλεύει η "`cin >> …`".***

4.5.1 Το Σύνολο Χαρακτήρων και οι Τύποι `char`

Η θέση του χαρακτήρα μέσα στο σύνολο χαρακτήρων τον καθορίζει απόλυτα. Π.χ. στη θέση 65 του ASCII υπάρχει το κεφαλαίο γράμμα '`A`' του λατινικού αλφαβήτου.

Οι γλώσσες προγραμματισμού βάζουν μερικές, πολύ χαλαρές, προδιαγραφές στα σύνολα χαρακτήρων που χρησιμοποιούν:

- Τα ψηφία '0', '1', ..., '9' είναι σε συναπτές θέσεις του πίνακα και κατ' αύξουσα τάξη.
- Αν τα πεζά γράμματα του Λατινικού αλφαβήτου υπάρχουν στον πίνακα, τότε είναι αλφαβητικά διαταγμένα, αλλά όχι αναγκαία σε συναπτές θέσεις.
- Αν τα κεφαλαία γράμματα του Λατινικού αλφαβήτου υπάρχουν στον πίνακα, τότε είναι αλφαβητικά διαταγμένα, αλλά όχι αναγκαία σε συναπτές θέσεις.

Αν έχουμε μια τιμή c τύπου `unsigned char`, με τυποθεώρηση με τη `static_cast<int>` μπορούμε να βρούμε τη θέση (order) της στο σύνολο χαρακτήρων. Π.χ. οι:

```
unsigned char c;

c = 'D';  cout << static_cast<int>(c);
```

```
c = 'Γ';  cout << "   " << int(c);
cout << "   "
 << static_cast<int>(static_cast<unsigned char>('ώ')) << endl;
```

θα δώσουν:

```
68   195    254
```

Αντιστρόφως, τυποθεώρηση τιμής τύπου `int` από τον `unsigned char` μας δίνει το χαρακτήρα που υπάρχει στη θέση που καθορίζει το όρισμα. Π.χ. οι:

```
cout << static_cast<unsigned char>(68) << "   ";
cout << static_cast<unsigned char>(195) << "   ";
cout << static_cast<unsigned char>(254) << endl;
```

θα δώσουν:

```
D   Γ   ώ
```

Μπορούμε λοιπόν να πούμε ότι: για οποιαδήποτε τιμή *c*, τύπου `unsigned char` έχουμε:

`static_cast<unsigned char>(static_cast<int>(c)) == c`

και αντιστρόφως (`int k`, $0 \le$ `k` ≤ 255):

`static_cast<int>(static_cast<unsigned char>(k)) == k`

Αλλά, πώς αποθηκεύεται στη μνήμη μια τιμή τύπου `unsigned char`; Αποθηκεύεται η «ζωγραφιά»; Όχι βέβαια. Μετά την εκτέλεση της εντολής: `c = 'D'` αυτό που θα αποθηκευτεί στη μνήμη, στη θέση *c*, είναι ο *ακέραιος* που δίνεται από την `static_cast<int>('D')`, σε δυαδική παράσταση.

Το να εμφανιστεί η ζωγραφιά στην οθόνη σου ή στον εκτυπωτή σου είναι ένα άλλο πρόβλημα που έχει να κάνει με το μέσο και την τεχνολογία του. Παλιότερα, στον τύπο `unsigned char` ήταν δυνατή η παράσταση όλων των χαρακτήρων που μπορούσε να εμφανίσει ένας ΗΥ. Σήμερα τα πράγματα είναι πιο πολύπλοκα. Οι ΗΥ έχουν δυνατότητα παράστασης πάρα πολλών χαρακτήρων –ο κατάλληλος τύπος είναι πια ο `wchar_t` που θα δούμε στη συνέχεια– ενώ ο `unsigned char` παριστάνει ένα επιλεγμένο υποσύνολο που μπορεί να αλλάζει.

Οι τύποι `signed char` και `char` έχουν και αυτοί τη δυνατότητα να παραστήσουν όλους τους χαρακτήρες που μπορεί να παραστήσει ο `unsigned char`, αλλά έχουμε μια διαφορά με το αποτέλεσμα της `int` για τέτοιες τιμές: αν το 8ο (ή το 1ο αν προτιμάς) δυαδικό ψηφίο είναι 1 ερμηνεύεται ως πρόσημο (–). Π.χ. οι:

```
unsigned char uc;
char c;

c = 'ώ';  uc = 'ώ';
cout << c << "   " << uc << endl;
cout << static_cast<int>(c) << "   "
     << static_cast<int>(uc) << endl;
c = c - 1;  uc = c;
cout << c << "   " << uc << endl;
cout << static_cast<int>(c) << "   "
     << static_cast<int>(uc) << endl;
```

θα δώσουν:

```
ώ   ώ
-2   254
ύ   ύ
-3   253
```

Ας δούμε τι ενδιαφέροντα υπάρχουν εδώ:

1. Και στις δύο μεταβλητές δώσαμε την ίδια τιμή, `'ώ'`, επιτυχώς, όπως φαίνεται και από την εκτύπωση των τιμών τους.
2. Οι `static_cast<int>(c)` και `static_cast<int>(uc)` προκάλεσαν την εκτύπωση των –2 και 254. Τι συνέβη; Πρόκειται για δύο διαφορετικές ερμηνείες της ίδιας δυαδικής παράστασης: 11111110_2. Στην πρώτη περίπτωση, που περιμένουμε προσημασμένο

αριθμό, το 1ο "1" θεωρείται ότι δείχνει αρνητική τιμή. Αυτά θα τα μάθουμε αργότερα. Πάντως προς το παρόν μπορείς να θυμάσαι τον εξής κανόνα:

Έστω ότι οι *c* (`int`) και *uc* (`unsigned int`) έχουν ίδια τιμή (στην περίπτωση αυτή: τον ίδιο χαρακτήρα)

αν `static_cast<int>(c)` ≥ 0 τότε
`static_cast<int>(c) == static_cast<int>(uc)`
αλλιώς (αν `static_cast<int>(c)` < 0)
`static_cast<int>(c) == (static_cast<int>(uc) - 256)`

3. Μειώσαμε την τιμή της *c* με τη `c = c - 1`. Έχουμε αυτό το δικαίωμα, αφού οι `char` θεωρούνται ακέραιοι τύποι.
4. Εκχωρήσαμε την τιμή της *c* στη *uc*· μην ξεχνάς ότι το νόημα της εκχώρησης είναι `uc = static_cast<unsigned char>(c)`. Μετά την εκχώρηση οι δύο μεταβλητές έχουν το ίδιο περιεχόμενο (`'ϋ'`).

Στα Windows, στους τύπους `char` και `signed char` όλοι οι χαρακτήρες παριστάνονται με ακέραιες τιμές από −128 μέχρι 127.

4.6 Ο Τύπος char στο Πρόγραμμα

Ας δούμε τώρα το εξής πρόβλημα: Σε κάποιο πρόγραμμά σου έχεις δηλώσει:

```
int k;
```

και στη συνέχεια, την εντολή:

```
cin >> k;
```

Όταν η εντολή αυτή εκτελείται, ας πούμε ότι θέλεις να δώσεις την τιμή 375. Πιέζεις λοιπόν τα πλήκτρα <3>, το <7> και το <5>. Όπως ήδη ξέρεις, το *k* θα πάρει την τιμή 375, αλλά ας δούμε πώς.

Αυτό που «φεύγει» από το πληκτρολόγιο προς τον υπολογιστή είναι η πληροφορία ότι πιέσθηκε το πλήκτρο <3> και στη συνέχεια το <7> και τέλος το <5>. Μέχρι εδώ λοιπόν το πρόγραμμά σου έχει «παραλάβει» τρεις *χαρακτήρες*, τρεις τιμές τύπου `char`. Πώς θα βγει τώρα το 375; Αυτό θα γίνει, με την εκτέλεση μερικών εντολών που παίρνουν ως στοιχεία εισόδου τους τρεις χαρακτήρες και βγάζουν ως αποτέλεσμα την τιμή 375 στη θέση μνήμης *k* του προγράμματός μας. Στη συνέχεια με ένα απλοϊκό προγραμματάκι θα προσπαθήσουμε να σου δείξουμε πώς γίνεται αυτή η δουλειά.

Κατ' αρχάς να επαναλάβουμε τον περιορισμό που είπαμε για τα σύνολα χαρακτήρων: «Τα ψηφία '0', '1', ..., '9' είναι σε συναπτές θέσεις του πίνακα και κατ' αύξουσα τάξη». Π.χ. στον ASCII οι θέσεις αυτές είναι από το 48 (`'0'`) μέχρι το 57 (`'9'`). Έτσι, αν από τη σειρά ενός ψηφίου στον πίνακα αφαιρέσουμε την σειρά που έχει ο χαρακτήρας '0' παίρνουμε την αριθμητική τιμή του ψηφίου αυτού. Π.χ. (στο ASCII):

`static_cast<int>('7')-static_cast<int>('0')` = 55 − 48 = 7

Με βάση τις παραπάνω παρατηρήσεις, μπορούμε να δούμε πώς περίπου διαβάζει τα αριθμητικά στοιχεία η C++. Ας πούμε ότι θέλουμε να γράψουμε ένα πρόγραμμα που θα διαβάζει τριψήφιους ακέραιους χωρίς πρόσημο· ή καλύτερα, θα διαβάζει τρεις χαρακτήρες - ψηφία και θα τους μεταφράζει σε ακέραιο αριθμό. Από τα τρία ψηφία: το πρώτο θα είναι το ψηφίο των εκατοντάδων, το δεύτερο των δεκάδων και το τρίτο των μονάδων. Αφού υπολογίσουμε την τιμή που αντιστοιχεί σε κάθε ψηφίο, τις προσθέτουμε πολλαπλασιασμένες με την αντίστοιχη δύναμη του 10. Να το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   const int ord0 = static_cast<int>('0');
// η θέση του '0' στο σύνολο χαρακτήρων
```

```
   unsigned char  digit1, digit2, digit3;
// τρία ψηφία (χαρακτήρες) που διαβάζονται από το πληκτρολόγιο
   unsigned char  ceol;  // εδώ θα πάει το <enter>
   int value1, value2, value3;
// οι ακέραιοι που αντιστοιχούν στα τρία ψηφία
   int integer;
// Ο ακέραιος που φιλοδοξούμε να διαβάσουμε

   cin.get(digit1);  cin.get(digit2);  cin.get(digit3);
   cin.get(ceol);
   value1 = static_cast<int>(digit1) - ord0;
   value2 = static_cast<int>(digit2) - ord0;
   value3 = static_cast<int>(digit3) - ord0;
   integer = value1*100 + value2*10 + value3;
   cout << " Ακέραιος = " << integer << endl;
}
```

Αν δώσουμε σ' αυτό το πρόγραμμα:

375

θα πάρουμε απάντηση:

```
 Ακέραιος = 375
```

Αν δώσουμε:

007

θα πάρουμε:

```
 Ακέραιος = 7
```

Το πρόγραμμα περιμένει **τρία δεκαδικά ψηφία.** Με διαφορετικά στοιχεία εισόδου δεν θα πάρεις την απάντηση που περιμένεις. Αν δώσουμε:

␣␣*7*

θα πάρουμε λάθος απάντηση:

```
 Ακέραιος = -1753
```

Και η εκτύπωση πώς γίνεται; Η οθόνη σου ή ο εκτυπωτής σου περιμένουν χαρακτήρες. Αν θέλουμε να γράψουμε τον αριθμό 375 θα πρέπει πρώτα να γράψουμε το ψηφίο των εκατοντάδων (`'3'`), μετά το ψηφίο των δεκάδων (`'7'`) και τέλος το ψηφίο των μονάδων (`'5'`).

Οι εκατοντάδες υπολογίζονται ως πηλίκο της *ακέραιης* διαίρεσης 375 `/` 100. Αν πάρουμε το υπόλοιπο αυτής της διαίρεσης (375 `%` 100 `==` 75) και το διαιρέσουμε δια 10, το πηλίκο (75 `/` 10) μας δίνει τις δεκάδες, ενώ το υπόλοιπο (75 `%` 10) μας δίνει τις μονάδες.

Οι παρακάτω εντολές κάνουν αυτή τη δουλειά για τριψήφιους μη-αρνητικούς ακέραιους.

```
   cin >> integer;
   value1 = integer / 100;
   digit1 = static_cast<unsigned char>(ord0 + value1);
   integer = integer % 100;  value2 = integer / 10;
   digit2 = static_cast<unsigned char>(ord0 + value2);
   integer = integer % 10;  value3 = integer;
   digit3 = static_cast<unsigned char>(ord0 + value3);
   cout << digit1 << digit2 << digit3 << endl;
```

Η C++ σου δίνει συναρτήσεις για επεξεργασία τιμών τύπου `char`. Στον Πίν. 4-3 βλέπεις μερικές από αυτές, που σου επιτρέπουν να δεις τι είδους είναι κάποιος χαρακτήρας.

Οι συναρτήσεις επιστρέφουν τιμή τύπου `int` για συμβατότητα με τη C, από την οποίαν και κληρονομήθηκαν. Θα τις χειριζόμαστε ως συναρτήσεις τύπου `bool` (κατηγορήματα). Όταν τις χρησιμοποιείς θα πρέπει να βάζεις στο πρόγραμμά σου την οδηγία: "`#include <cctype>`". Ας δούμε ένα παράδειγμα. Το πρόγραμμα:

```
#include <iostream>
#include <cctype>
```

Όνομα	Δίνει αποτέλεσμα `true` (≠ 0) αν το όρισμα είναι:
`isalpha`	Λατινικό γράμμα (`'a'..'z'`, `'A'..'Z'`)
`isupper`	Κεφαλαίο λατινικό γράμμα (`'A'..'Z'`)
`islower`	Πεζό λατινικό γράμμα (`'a'..'z'`)
`isdigit`	Δεκαδικό ψηφίο (`'0'..'9'`)
`isxdigit`	Δεκαεξαδικό ψηφίο (`'0'..'9'`, `'a'..'f'`, `'A'..'F'`)
`isspace`	Κάποιος από τους `' '`, `'\t'`, `'\r'`, `'\n'`, `'\f'`
`iscntrl`	Χαρακτήρας ελέγχου: `static_cast<char>(0) .. static_cast<char>(31)`, `static_cast<char>(127)`
`ispunct`	Τίποτε από τα παραπάνω
`isalnum`	Λατινικό γράμμα ή δεκαδικό ψηφίο
`isprint`	Εκτυπώσιμος: `static_cast<char>(32)` (= `' '`) .. `static_cast<char>(126)` (= `'~'`)
`isgraph`	`isalpha || isdigit || ispunct`
`isascii`	Χαρακτήρας ASCII (ISO 646) `static_cast<char>(0) .. static_cast<char>(127)`

Πίν. 4-3 Συναρτήσεις της C++ για επεξεργασία χαρακτήρων. Παίρνουν ένα όρισμα τύπου `char`. Επιστρέφουν τιμή `true` αν το όρισμα έχει την ιδιότητα που περιγράφεται στη δεύτερη στήλη. Αλλιώς `false`. Για να τις χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου: `#include <ctype>`.

```
using namespace std;
int main()
{
   char c1 = 'f', c2 = 'ϕ', c3 = '3';

   cout << isalpha(c1) << "  " << isalpha(c2) << endl;
   cout << (isxdigit(c1) && !isdigit(c1)) << endl;
   cout << (isxdigit(c3) && !isdigit(c3)) << endl;
}
```

μας δίνει:

```
8  0
1
0
```

Το `'f'` είναι γράμμα του λατινικού αλφαβήτου και η `isalpha(c1)` μας δίνει τιμή `8` ≠ 0 δηλαδή `true`. Το `'ϕ'` δεν είναι γράμμα του λατινικού αλφαβήτου· η `isalpha(c2)` μας δίνει τιμή `0` δηλαδή `false`. Στη συνέχεια ζητάμε την τιμή της πρότασης «το `'f'` είναι δεκαεξαδικό ψηφίο και το `'f'` δεν είναι δεκαδικό ψηφίο». Η απάντηση είναι `true` (1). Αλλά, στη συνέχεια μας λέει ότι η τιμή της πρότασης «το `'3'` είναι δεκαεξαδικό ψηφίο και το `'3'` δεν είναι δεκαδικό ψηφίο» είναι `false` (0).

Α, και κάτι ακόμη: αν σε ενοχλεί το 8 άλλαξε την πρώτη εντολή σε:

```
   cout << static_cast<bool>(isalpha(c1)) << "  "
        << static_cast<bool>(isalpha(c2)) << endl;
```

Πάντως, οι συνθήκες με τις λογικές πράξεις υπολογίζονται σωστά· δεν υπάρχει πρόβλημα.

Στο `cctype` δηλώνεται ακόμη η συνάρτηση *tolower* που αν τροφοδοτηθεί με κεφαλαίο γράμμα του λατινικού αλφαβήτου μας δίνει το αντίστοιχο πεζό, π.χ. η `tolower('Q')` θα δώσει `'q'`. Η *toupper*, που δηλώνεται επίσης στο `cctype`, αν τροφοδοτηθεί με πεζό θα μας δώσει το αντίστοιχο κεφαλαίο, π.χ. η `toupper('q')` θα δώσει `'Q'`.

4.7 Ο Τύπος wchar_t

Ο τύπος **wchar_t** είναι ένα εργαλείο της C++[5] που χρησιμοποιείται για να αντιμετωπισθεί το πρόβλημα της παράστασης στον υπολογιστή πολλών χαρακτήρων. Έχει 65536 διαφορετικές τιμές, αντί για τις 256 του **char** και χρησιμοποιείται για την υλοποίηση του προτύπου Unicode.

Μια σταθερά τύπου **wchar_t** είναι μια σταθερά τύπου **char** με το πρόθεμα **L**, π.χ.:

```
L'a'   L'%'  L'ξ'
```

Όπως λέγαμε και στον Πίν. 2-1, υλοποιείται σε 16 δυαδικά ψηφία (2 ψηφιολέξεις). Μπορείς να δοκιμάσεις την:

```
cout << sizeof('A') << "  " << sizeof(L'A') << endl;
```

που θα σου δώσει:

```
1  2
```

Δεν μπορείς να διαβάσεις τιμές τύπου **wchar_t** από το *cin* ενώ μπορείς να στείλεις τιμές στο ρεύμα *cout*, αλλά θα δεις να τυπώνονται οι αντίστοιχοι αριθμοί. Π.χ. αν

```
wchar_t a, b;
```

τότε οι

```
a = L'a';  b = L'\n';
cout << a << "  " << b << endl;
```

θα δώσουν:

```
97  10
```

Αν περιλάβεις στο πρόγραμμά σου το **cwctype**[6] μπορείς να χρησιμοποιήσεις τις συναρτήσεις επεξεργασίας τιμών τύπου **wchar_t**, αντίστοιχες αυτών που έχουμε στον Πίν. 4-3. Τα ονόματά τους διαφέρουν κατά το ότι αντί για **is** έχουν πρόθεμα **isw**. Π.χ., για τις παραπάνω τιμές των *a*, *b*, η

```
cout << iswalnum(a) << "  " << iswprint(b)  << endl;
```

θα δώσει:

```
256  0
```

(ο **L'a'** είναι αλαφαριθμητικός ενώ ο **L'\n'** δεν είναι εκτυπώσιμος.)

Παρατήρηση: ▶

Σύμφωνα με αυτά που είπαμε, στην *a* μπορούμε να αποθηκεύσουμε το ελληνικό γράμμα "πεζό άλφα" στην κωδικοποίηση Unicode, που είναι η τιμή **0x03b1** (945) γράφοντας "**a = 0x03b1**" ή "**a = 945**". Όχι όμως γράφοντας "**a = L'α'**"· στην περίπτωση αυτή αποθηκεύεται η τιμή 225 (Windows).◀

4.8 Τακτικοί Τύποι

Ξεκινούμε με την εξής παρατήρηση:

- Μπορούμε να απεικονίσουμε τις τιμές του τύπου **bool** στο υποσύνολο { 0, 1 } του **int**.
- Μπορούμε να απεικονίσουμε τις τιμές του τύπου **unsigned char** στο υποσύνολο 0 .. 255 του **int**.
- Μπορούμε να απεικονίσουμε τις τιμές των τύπων **signed char** και **char** στο υποσύνολο −128 .. 127 του **int**.

---

[5] Στη C ορίζεται στο stddef.h ως (Borland C):
```
typedef unsigned short wchar_t;
```

[6] Αν δεν το έχεις δοκίμασε το **cctype**.

Η απεικόνιση –και στις τρεις περιπτώσεις– γίνεται με την `int`. Οι τύποι αυτοί λέγονται **τακτικοί τύποι** (ordinal types)[7]. Φυσικά και ο `int`, που μπορεί να απεικονισθεί στον εαυτό του, είναι επίσης τακτικός τύπος. Και στην περίπτωση αυτή η απεικόνιση γίνεται με την `int`, αλλά, φυσικά για τον `int`, η `int` απεικονίζει κάθε αριθμό στον εαυτό του.

Οι τακτικοί τύποι είναι **διαταγμένοι** (ordered). Για κάθε τιμή ενός τακτικού τύπου υπάρχει μια **προηγούμενη** (predecessor) και μια **επόμενη** (successor). Π.χ. προηγούμενος του 0 είναι ο –1 και επόμενος ο 1, προηγούμενος του `'C'` είναι ο `'B'` και επόμενος ο `'D'`. Φυσικά, δεν υπάρχει επόμενος του *INT_MAX* και προηγούμενος του *INT_MIN*. Στον τύπο `char` δεν υπάρχει προηγούμενος του `char(-128)` και επόμενος του `char(127)`. Στον `bool` δεν υπάρχει προηγούμενη της `false` ούτε επόμενη της `true`.

4.9 Απαριθμητοί Τύποι (που Ορίζονται από τον Χρήστη)

Η C++ δίνει τη δυνατότητα στον προγραμματιστή να ορίζει δικούς του τύπους. Μια τέτοια κατηγορία τύπων είναι και τακτικοί ή **απαριθμητοί τύποι** (enumerated types). Μπορείς, για παράδειγμα, στο πρόγραμμά σου να ορίσεις:

```
   enum WeekDay { sunday, monday, tuesday, wednesday, thursday,
                  friday, saturday };
   enum EurUn { Austria, Belgium, Bulgaria, Cyprus, CzechRepublic,
                Denmark, Ellas, Estonia, Finland, France, Germany,
                Hungary, Ireland, Italy, Latvia, Lithuania,
                Luxembourg, Malta, Netherlands, Poland, Portugal,
                Romania, Slovakia, Slovenia, Spain, Sweden,
                UnitedKingdom };
```

στη συνέχεια να δηλώσεις:

```
   WeekDay day1, day2;
   EurUn country;
```

και να δώσεις τιμές:

```
   day1 = sunday;  day2 = wednesday;
// . . .
   country = Ellas;
```

Ο ορισμός ενός απαριθμητού τύπου αρχίζει με το λεξικό σύμβολο "`enum`". Στη συνέχεια γράφουμε το όνομα του τύπου και μετά, μέσα σε άγκιστρα, τα *ονόματα των τιμών* που περιλαμβάνονται στον τύπο.

Φυσικά, ο ορισμός τύπου μπαίνει, στο πρόγραμμά μας, *πριν* από τις δηλώσεις μεταβλητών.

Ο ορισμός του τύπου είναι συγχρόνως και ορισμός των τιμών που περιέχει. Για παράδειγμα, οι τιμές που μπορεί να πάρει κάθε μεταβλητή τύπου *WeekDay*, όπως η *day*1, είναι οι: *sunday*, *monday*, *tuesday*, *wednesday*, *thursday*, *friday*, *saturday*.

Οι τιμές του κάθε τύπου είναι διαταγμένες σύμφωνα με τον ορισμό του τύπου. Π.χ.:

sunday < *monday* < *tuesday* < *wednesday* < *thursday* < *friday* < *saturday*

Austria < *Belgium* < *Bulgaria* < . . . < *Sweden* < *UnitedKingdom*

Η `static_cast<int>` δέχεται ως όρισμα τιμή τέτοιων τακτικών τύπων και μας δίνει ως τιμή τη θέση (τάξη) του ορίσματος στον τύπο:

`static_cast<int>(sunday) == 0`,
`static_cast<int>(monday) == 1`,
`static_cast<int>(tuesday) == 2` κ.ο.κ.
`static_cast<int>(Austria) == 0`,
`static_cast<int>(Belgium) == 1`,
`static_cast<int>(Denmark) == 2` κ.ο.κ.

[7] Ο όρος από την Pascal.

Μαζί με κάθε τέτοιο τύπο ορίζεται και η αντίστοιχη αντίστροφη συνάρτηση που μας δίνει τιμές του τύπου αυτού:

`static_cast<WeekDay>(0) == sunday`,
`static_cast<WeekDay>(1) == monday`,
`static_cast<WeekDay>(2) == tuesday` κ.ο.κ.
`static_cast<EurUn>(0) == Austria`,
`static_cast<EurUn>(1) == Belgium`,
`static_cast<EurUn>(2) == Bulgaria` κ.ο.κ.

Μπορείς να χρησιμοποιείς τέτοιες τιμές σε εντολές εξόδου· η "`cout << v …`" θα λειτουργήσει σαν την: "`cout << static_cast<int>(v)...`"

Δεν μπορείς να χρησιμοποιείς μεταβλητές τέτοιων τύπων σε εντολές εισόδου (`cin >> v`). Το πρόβλημα παρακάμπτεται με τη χρήση μιας μεταβλητής "`int k`" και με τις εξής εντολές:

```
cin >> k;
v = static_cast<T>( k );
```

όπου *T* ο τύπος της *v*.

Μπορείς αν θέλεις να αλλάζεις τις τιμές του `int` στους οποίους απεικονίζονται οι τιμές του τύπου σου. Π.χ. με τον ορισμό:

```
enum DecDigit { zero = 48, one, two, three, four, five, six,
                seven, eight, nine };
```

θα έχεις:

```
static_cast<int>( zero) == 48,
static_cast<int>(one) == 49,
. . .
static_cast<int>(nine) == 57
```

Αν θέλεις μπορείς να δίνεις περισσότερα από ένα ονόματα στην ίδια τιμή. Με τους παρακάτω ορισμούς:

```
enum Digit { miden = 48, zero = 48, one, two, three, four,
             five, six, seven, eight, nine };
enum Digit { zero = 48, one, two, three, four,
             five, six, seven, eight, nine, miden = 48 };
```

θα έχεις:

```
static_cast<int>(miden) == static_cast<int>(zero) ==48,
static_cast<int>(one) == 49,
. . .
static_cast<int>(nine) == 57
```

Αλλά προσοχή· αν δώσεις:

```
enum Digit { zero = 48, one, two, three, four,
             miden = 48, five, six, seven, eight, nine };
```

θα έχεις:

```
static_cast<int>(miden) == static_cast<int>(zero) == 48,
static_cast<int>(one) == static_cast<int>(five) == 49,
static_cast<int>(two) == static_cast<int>(six) == 50,
. . .
```

4.10 Μετονομασία Τύπου

Θα μπορούσαμε να ορίσουμε τους τύπους *WeekDay* και *EurUn* και ως εξής:

```
typedef enum { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday }                        WeekDay;
typedef enum { Austria, Belgium, Bulgaria, Cyprus,
               CzechRepublic, Denmark, Ellas, Estonia, Finland,
               France, Germany, Hungary, Ireland, Italy,
               Latvia, Lithuania, Luxembourg, Malta,
```

```
                Netherlands, Poland, Portugal, Romania,
                Slovakia, Slovenia, Spain, Sweden,
                UnitedKingdom }                          EurUn;
```

Το **typedef** (*type def*inition) είναι λεξικό σύμβολο. Αν σκεφτούμε ότι ο τύπος είναι στην παργαματικότητα το **enum { sunday, monday, tuesday, wednesday, thursday, friday, saturday }**, η **typedef** είναι στην πραγματικότητα εντολή *μετονομασίας* τύπου. Με αυτήν μπορείς να μετονομάσεις και άλλους τύπους που είναι ήδη ορισμένοι. Π.χ.

```
typedef unsigned int       Natural;
typedef unsigned long int  LongNatural;
typedef unsigned char      byte;
typedef bool               Logical;
```

Οι «νέοι» τύποι, *Natural*, *LongNatural*, *Logical* και *byte*, είναι στην πραγματικότητα οι αρχικοί τύποι και, επομένως, έχουν όλες τις ιδιότητές τους. Μετά τους παραπάνω ορισμούς μπορείς να δηλώσεις:

```
Natural     n, eN;
LongNatural athr, aAth;
Logical     yparxei;
byte        c1, c2;
```

Ασκήσεις

Α Ομάδα

4-1 Εξήγησε πώς βγήκε το αποτέλεσμα –1753 όταν δώσαμε στο πρόγραμμα της §4.5, στοιχείο εισόδου: ␣␣**7**.

Β Ομάδα

4-2 Γράψε πρόγραμμα που θα αποδεικνύει την ισοδυναμία: ((*A* **||** *B*) **&&** *B*) ≡ *B*.

4-3 Το ίδιο για τις ταυτολογίες: (*A* **&&** *B*) ⇒ *A* και (*A* **&&** *B*) ⇒ *B*.

Γ Ομάδα

4-4 Με βάση το πρόγραμμα της §4.5, γράψε πρόγραμμα που θα διαβάζει *δυαδικούς ακέραιους*.