κεφάλαιο

17

* Εσωτερική Παράσταση Δεδομένων

Ο στόχος μας σε αυτό το κεφάλαιο:
Να μάθεις
- πώς παριστάνονται στη μνήμη του ΗΥ οι τιμές διαφόρων τύπων και
- περιορισμούς και προβλήματα που προκύπτουν από τους τρόπους παράστασης.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να γράφεις πιο αξιόπιστα προγράμματα.

Έννοιες κλειδιά:
- *παράσταση ακεραίων*
- *παράσταση πραγματικών*
- *διαχείριση δυαδικών ψηφίων*
- *ψηφιοπράξεις*
- *σφάλματα παράστασης*
- *σφάλματα πράξεων*

Περιεχόμενα:

17.1 Παράσταση Φυσικών 541
17.2 Παράσταση Ακεραίων - Αρνητικοί Αριθμοί 543
17.2.1 * Ακέραιοι Τύποι του C99 544
17.3 Οι Ακέραιοι στο Πρόγραμμα 545
17.3.1 Παράμετροι “unsigned” 547
17.4 * Απαριθμητοί Τύποι (ξανά) 548
17.5 Ψηφιοπράξεις στη C++ 549
17.6 Ψηφιοχάρτες και Συνηθισμένες Πράξεις 552
17.6.1 Τιμή Δυαδικού Ψηφίου 553
17.6.2 Βάλε Τιμή 1 σε Δυαδικό Ψηφίο 554
17.6.3 Βάλε Τιμή 0 σε Δυαδικό Ψηφίο 555
17.6.4 Πλήθος "1" 556
17.6.5 Μέρος Ψηφιοχάρτη 556
17.7 Τύποι *bitmask* 557
17.8 Αριθμητικές Πράξεις και Ψηφιοπράξεις 560
17.9 Παράσταση και Πράξεις στον Τύπο “float” 560
17.9.1 Υπολογισμός Περιοδικής Συνάρτησης 564
17.10 Άλλοι Τύποι Κινητής Υποδιαστολής 564
17.11 Ο Τύπος “float” στο Δυαδικό Σύστημα 565
17.11.1 Πόλωση 566
17.11.2 Άλλες Περιπλοκές - Πρότυπο IEEE 567
17.11.3 Οι Τύποι “double” και “long double” 568
17.11.4 Μερικά Χρήσιμα Εργαλεία από τη C 569
17.12 Σφάλμα από Μετατροπή Τύπου 570

17.13 Τα Σφάλματα και πώς Μεταδίδονται ... 570
17.13.1 Το Σφάλμα Παράστασης ... 571
17.13.2 Μετάδοση Σφαλμάτων ... 571
17.14 Ισότητα στους Τύπους Κινητής Υποδιαστολής ... 573
17.15 Πρακτικές Συμβουλές ... 576
Ασκήσεις ... 578
Α Ομάδα ... 578
Β Ομάδα ... 579
Γ Ομάδα ... 579

Εισαγωγικές Παρατηρήσεις – Αριθμητικά Συστήματα:

Ένας από τους πιό γνωστούς τρόπους κωδικοποίησης αριθμητικών πληροφοριών είναι η κωδικοποίηση στο γνωστό **δεκαδικό σύστημα**, στο οποίο χρησιμοποιούμε δέκα διαφορετικά σύμβολα, τα ψηφία: 0,1,2,3,4,5,6,7,8,9.

Για να παραστήσουμε στο σύστημα αυτό έναν ακέραιο αριθμό (θετικό ή αρνητικό), χρησιμοποιούμε ένδεκα σύμβολα: τα δέκα ψηφία και το πρόσημο "-" (μείον). Πολλές φορές χρησιμοποιούμε και το πρόσημο "+" (συν), για να ξεχωρίζουμε ευκολότερα τους θετικούς αριθμούς από τους αρνητικούς (π.χ. -5109, 2048, +2048). Ακόμη, στην παράσταση ενός κλασματικού αριθμού χρειαζόμαστε και την υποδιαστολή (ευρωπαϊκές χώρες: ",", ΗΠΑ. Μεγ. Βρεττανία: "."), που διαχωρίζει το ακέραιο μέρος του αριθμού από το κλασματικό (π.χ. 48,25).

Το κοινό δεκαδικό αριθμητικό σύστημα είναι **θεσιακό** (positional), διότι η τιμή που παριστάνει κάθε ψηφίο ενός αριθμού εξαρτάται από τη θέση που έχει στην παράσταση του αριθμού. Για παράδειγμα, το πρώτο ψηφίο 3 στον αριθμό 6343 σημαίνει την τιμή 300 (3×10^2), ενώ το δεύτερο ψηφίο 3 την τιμή 3 (3×10^0).

Εκτός από τα θεσιακά συστήματα αριθμήσεως υπάρχουν και μη θεσιακά συστήματα, όπως είναι το γνωστό –**προσθετικό** (additive)– ρωμαϊκό σύστημα (π.χ. XXXIII = 10 + 10 + 10 + 1 + 1 + 1 = 33) ή το Ελληνικό (ρβ' = 100 + 2 = 102), όπου τα σύμβολα που χρησιμοποιούμε έχουν την ίδια τιμή ανεξαρτήτως της θέσης τους στον αριθμό που παριστάνουν.

Σε ένα θεσιακό αριθμητικό σύστημα κάθε φυσικός αριθμός N μπορεί να παρασταθεί με ένα πολυώνυμο της εξής μορφής:

$$N = \psi_{L-1}B^{L-1} + \psi_{L-2}B^{L-2} + \ldots + \psi_0 B^0 \quad (1)$$

Ο αριθμός B λέγεται **βάση** (base, radix) του αριθμητικού συστήματος και υποθέτουμε γενικώς ότι είναι μεγαλύτερος από το 1 (υπάρχουν όμως και συστήματα με αρνητική βάση). Οι συντελεστές ψ_k (k: 0..L-1) που μπορεί να πάρουν τιμές στο [0..B) ($0 \le \psi_k < B$), αποτελούν τα διαδοχικά ψηφία του αριθμού N ο οποίος έχει L θέσεις (ή ψηφία). Έτσι, η βάση B καθορίζει το πλήθος των διαφορετικών ψηφίων ενός αριθμητικού συστήματος. Στο δεκαδικό αριθμητικό σύστημα έχουμε βάση $B = 10$ και τα δέκα διαφορετικά ψηφία είναι: 0, 1, 2,...,9.

Δηλαδή επιλέγοντας τη βάση, δημιουργούμε ένα αριθμητικό σύστημα. Όταν π.χ. η βάση B = 2 ή 3, 8, 10, 16, τότε το σύστημα λέγεται **δυαδικό** ή αντιστοίχως **τριαδικό, οκταδικό, δεκαδικό, δεκαεξαδικό**. Έτσι λοιπόν ο αριθμός N = 7635, του δεκαδικού θεσιακού συστήματος, μπορεί να γραφτεί με την εξής μορφή:

$$N = 7\times10^3 + 6\times10^2 + 3\times10^1 + 5\times10^0 = 7635$$

όπου $\psi_3 = 7$, $\psi_2 = 6$, $\psi_1 = 3$ και $\psi_0 = 5$.

Στο οκταδικό σύστημα η βάση $B = 8$ και τα διάφορα ψηφία αυτού του συστήματος είναι: 0,1,2,...,7. Επομένως ο αριθμός N = 7635, του οκταδικού συστήματος, παριστάνει την τιμή του δεκαδικού αριθμού 5149, επειδή:

$$N = 7635_8 = 7\times8^3 + 6\times8^2 + 3\times8^1 + 5\times8^0 = 5149_{10}$$

Βλέπουμε λοιπόν ότι η ίδια ακολουθία ψηφίων (π.χ. 7635) αντιπροσωπεύει άλλη τιμή στο οκταδικό σύστημα και άλλη στο δεκαδικό. Έτσι, για να μη γίνεται σύγχυση, συχνά γράφουμε σαν κάτω δείκτη, στο δεξιό μέρος της ακολουθίας ψηφίων ενός αριθμού τη βάση του αριθμητικού συστήματος που χρησιμοποιούμε (πάντα στο δεκαδικό συμβολισμό). Στην

παρουσίαση των αριθμών στα θεσιακά συστήματα σημειώνουμε, για λόγους απλοποίησης, μόνο τους όρους ψ_k (σε κατιούσα σειρά) και παραλείπουμε τις δυνάμεις *B*.

17.1 Παράσταση Φυσικών

Στους ψηφιακούς ΗΥ χρησιμοποιείται το **δυαδικό** (binary) *αριθμητικό σύστημα*, για λόγους τεχνολογικής αξιοπιστίας (αλλά και θεωρητικούς). Η Φυσική και η Ηλεκτρονική έχουν να προτείνουν **συστήματα δύο καταστάσεων** (two state systems), στα οποία μπορούμε

- να ανιχνεύουμε την κατάσταση που βρίσκεται το σύστημα και
- να το βάζουμε στην κατάσταση που θέλουμε.

Σε ένα τέτοιο σύστημα, «βαφτίζουμε» τη μια κατάσταση "0" και την άλλην "1"· παριστάνουμε δηλαδή τα δύο ψηφία του δυαδικού συστήματος ($B = 2$).

Σύμφωνα με αυτά που είπαμε παραπάνω, στο δυαδικό σύστημα κάθε φυσικός αριθμός παριστάνεται ως άθροισμα δυνάμεων του 2. Για παράδειγμα, ο δεκαδικός αριθμός $N = 12$ ($8 + 4 = 2^3 + 2^2$), μπορεί να παρασταθεί ως εξής:

$$N = 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 = 1100_2$$

όπου $\psi_3 = 1$, $\psi_2 = 1$, $\psi_1 = 0$ και $\psi_0 = 0$.

Συχνά στα υπολογιστικά συστήματα χρησιμοποιείται και το **δεκαεξαδικό** (hexadecimal) αιθμητικό σύστημα, για το οποίο $B = 16$ και τα ψηφία του είναι:

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`

Τα νέα σύμβολα *A*, *B*, *C*, *D*, *E* και *F* (ή *a*, *b*, *c*, *d*, *e*, *f*) αντιστοιχούν στις τιμές των (δεκαδικών) αριθμών: δέκα, ένδεκα, δώδεκα, δεκατρία, δεκατέσσερα και δεκαπέντε. Κατά συνέπεια ο δεκαεξαδικός αριθμός 153 σημαίνει:

$$153_{16} = 1\times16^2 + 5\times16^1 + 3\times16^0 = 256 + 80 + 3 = 339_{10}$$

και ο δεκαεξαδικός αριθμός $1A3F$ σημαίνει:

$$\begin{aligned} 1A3F_{16} &= 1\times16^3 + A\times16^2 + 3\times16^1 + F\times16^0 \\ &= 1\times16^3 + 10\times16^2 + 3\times16^1 + 15\times16^0 \\ &= 6719_{10} \end{aligned}$$

Ο Πίν. 17-1 δείχνει την παράσταση των ακεραίων αριθμών από 1 μέχρι 20 σε διάφορα θεσιακά αριθμητικά συστήματα.

Από τον Πίν. 17-1 φαίνεται ότι όταν μικραίνει η βάση του αριθμητικού συστήματος μεγαλώνει το πλήθος των ψηφίων που χρειάζονται για να παρασταθεί ένας αριθμός. Για παράδειγμα, ο αριθμός 9 χρειάζεται ένα ψηφίο στο 16-δικό και στο 10-δικό σύστημα, δυό στο 8-δικό, τρία στο 3-δικό και τέσσερα στο 2-δικό σύστημα.

Το πλήθος *Π* των διαφόρων αριθμών που μπορεί να γραφούν σε *L* θέσεις ενός αριθμητικού συστήματος βάσεως *B* δίνεται από τον τύπο:

$$\Pi = B^L \qquad (2)$$

Αν, ας πούμε, πάρουμε $B = 2$ και $L = 4$, τότε μπορούμε να ξεχωρίσουμε 16 ($=2^4$) διαφορετικούς δυαδικούς αριθμούς (ή συνδυασμούς), δηλ. τους αριθμούς: 0000, 0001, 0010,... 1111.

Αν τώρα γνωρίζουμε το πλήθος των ακεραίων αριθμών *Π*, που θέλουμε να παραστήσουμε στο αριθμητικό σύστημα με βάση το *B*, τότε ο απαιτούμενος αριθμός θέσεων *L* καθορίζεται από τον τύπο:

$$L = log_B\Pi \qquad (3)$$

Έτσι, για να παραστήσουμε τους πρώτους 16 φυσικούς αριθμούς, από 0 μέχρι 15, στο δυαδικό σύστημα χρειαζόμαστε $log_2 16 = 4$ θέσεις. Ενώ για την παράσταση των πρώτων 9 αριθμών στο τριαδικό σύστημα χρειαζόμαστε $log_3 9 = 2$ θέσεις. Μπορείς να ελέγξεις την ορθότητα του τύπου (3) και από τον Πίν. 17-1.

Ακόμη, από τον Πίν. 17-1, μπορείς να δεις ότι το οκταδικό και το δεκαεξαδικό είναι «συμπυκνώσεις» του δυαδικού συστήματος. Π.χ. το "20_{10}" στο δυαδικό γράφεται "10100". Αν

10δικό	16δικό	8δικό	3δικό	2δικό
0	00	00	000	00000
1	01	01	001	00001
2	02	02	002	00010
3	03	03	010	00011
4	04	04	011	00100
5	05	05	012	00101
6	06	06	020	00110
7	07	07	021	00111
8	08	10	022	01000
9	09	11	100	01001
10	0A	12	101	01010
11	0B	13	102	01011
12	0C	14	110	01100
13	0D	15	111	01101
14	0E	16	112	01110
15	0F	17	120	01111
16	10	20	121	10000
17	11	21	122	10001
18	12	22	200	10010
19	13	23	201	10011
20	14	24	202	10100

Πίν. 17-1 Παράσταση αριθμών σε διάφορα αριθμητικά συστήματα.

δεις αυτήν την παράσταση ως δυο τριάδες: "010 | 100" και κάθε μια από αυτές ως ψηφίο του οκταδικού συστήματος, παίρνεις τον "24_8". Παρομοίως, αν τη δεις ως δυο τετράδες "0001 | 0100" και τις γράψεις ως ψηφία του δεκαεξαδικού παίρνεις: "14_{16}". Φυσικά, αυτές οι ιδιότητες ξεκινούν από το ότι $8 = 2^3$ και $16 = 2^4$.

Στην §2.5 (Πίν. 2-1) είδαμε ότι η C++, για την παράσταση φυσικών αριθμών, μας δίνει τους ακέραιους τύπους χωρίς πρόσημο: `unsigned char`, `unsigned int`, `unsigned long`. Ο `unsigned char` αποθηκεύει τιμές σε μια ψηφιολέξη[1] των οκτώ δυαδικών ψηφίων· σύμφωνα με τον τύπο (2), μπορεί να παραστήσει $2^8 = 256$ διαφορετικές τιμές, τους φυσικούς από 0 μέχρι 255. Ο `unsigned short int` δουλεύει με μια λέξη (δυο ψηφιολέξεις) με 16 δυαδικά ψηφία και μπορεί να παραστήσει $2^{16} = 65\,536$ διαφορετικές τιμές, τους φυσικούς από 0 μέχρι 65 535. Τέλος, ο `unsigned long` μπορεί να παραστήσει $2^{32} = 4\,294\,967\,296$ τιμές, από 0 μέχρι 4 294 967 295. Και ο `unsigned int`; Μπορεί να σαν τον `unsigned long` ή σαν τον `unsigned short int`.

Στη συνέχεια θα δούμε έναν τρόπο για να «σκαλίζουμε» τις εσωτερικές παραστάσεις.

Στις §1.8.1 και §1.8.2 είδαμε ότι η C++ μας δίνει τη δυνατότητα να γράφουμε στο πρόγραμμά μας φυσικούς αριθμούς στο δεκαεξαδικό σύστημα βάζοντας το πρόθεμα "`0x`" ή "`0X`" και στο οκταδικό σύστημα βάζοντας το πρόθεμα "`0`". Π.χ., οι εντολές:

```
k = 255;   k = 0XFF;   k = 0xff;   k = 0377;
```

κάνουν ακριβώς το ίδιο πράγμα. Το "255" είναι μια αριθμητική σταθερά στο δεκαδικό σύστημα. Η ίδια τιμή στο δεκαεξαδικό σύστημα γράφεται "FF_{16}" και στο οκταδικό "377_8". Στη

[1] Οι τιμές των μεγεθών που δίνουμε εδώ δεν είναι υποχρεωτικές, είναι απλώς συνηθισμένες. Θυμίσου ότι όπως λέγαμε στο Κεφ. 2 το υποχρεωτικό είναι ότι:

`sizeof(char)` ≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`

C++, η δεκαεξαδική τιμή γράφεται "`0XFF`" και η οκταδική "`0377`". Πρόσεξε καλά την οκταδική γραφή:

♦ ***Όταν μια αριθμητική σταθερά ακέραιου τύπου αρχίζει με "`0`" (μηδέν) είναι οκταδικός και και όχι δεκαδικός αριθμός.***

17.2 Παράσταση Ακεραίων – Αρνητικοί Αριθμοί

Πώς μπορούμε να παραστήσουμε σε μια ψηφιολέξη ακέραιους θετικούς ή αρνητικούς; Αφού το πρόσημο μπορεί να είναι "`+`" ή "`-`", μπορούμε να το παραστήσουμε με ένα δυαδικό ψηφίο: "`0`" για το "`+`" και "`1`" για το "`-`". Έτσι, θα μας μείνουν άλλα 7 δυαδικά ψηφία για την απόλυτη τιμή, που θα μπορεί να είναι από 0 μέχρι $2^7 - 1 = 127$. Να λοιπόν ένας τρόπος για να παραστήσουμε ακέραιες τιμές από –127 μέχρι +127. Αυτός ο τρόπος παράστασης λέγεται «**πρόσημο - απόλυτη τιμή**». Αλλά πρόσεξε: εδώ έχουμε 255 τιμές, ενώ στον τύπο `unsigned char` παριστάνουμε 256 διαφορετικές τιμές από –128 μέχρι 127! Ναί, χάσαμε μια τιμή, διότι το 0 (μηδέν) παριστάνεται με δυο διαφορετικούς τρόπους: ως "`+0`" (`00000000`) και ως "`-0`" (`10000000`).

Συνήθως, στους υπολογιστές μας θα βρούμε την παράσταση αρνητικών με το "**συμπλήρωμα ως προς 2**" (2's complement). Ας δούμε ένα

Παράδειγμα ✍

Όπως είδαμε, το 12 παριστάνεται σε μια ψηφιολέξη, ως:

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

Για να βρούμε το συμπλήρωμα ως προς 2:

Βήμα 1: αλλάζουμε τα 0 σε 1 και τα 1 σε 0:

7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1

(αυτό είναι το **συμπλήρωμα ως προς 1**)

Βήμα 2: προσθέτουμε το 1

```
  1 1 1 1 0 0 1 1
+ 0 0 0 0 0 0 0 1
 -----------------
 |1|1|1|1|0|1|0|0|
  7 6 5 4 3 2 1 0
```

Αυτήν την παράσταση χρησιμοποιούμε για να παραστήσουμε το "`-12`".

✍✍✍

Το ότι η τιμή είναι αρνητική φαίνεται –όπως και στην «πρόσημο-απόλυτη τιμή»– από το "`1`" στο δυαδικό ψηφίο 7. Τί κερδίσαμε από όλα αυτά; Ας του προσθέσουμε την παράσταση του 20 (`00010100`) αγνοώντας το ότι το πρώτο ψηφίο είναι πρόσημο:

```
  11110100
+ 00010100
 ---------
 1|00001000
```

Αν ξεχάσουμε το πιο σημαντικό ψηφίο, που είναι "`1`", τα υπόλοιπα οκτώ ψηφία παριστάνουν το 8, που είναι σωστό: (-12) + 20 = 8.

Ένας άλλος τρόπος για να πούμε το ίδιο πράγμα είναι ο εξής: Το αποτέλεσμα της πράξης ήταν $2^8+2^3 = 264$. Κρατάμε το υπόλοιπο της διαίρεσής του δια 256 (= 2^8). Γι' αυτό, αυτή η αριθμητική λέγεται **αριθμητική υπολοίπων** (modulo arithmetic).

Δηλαδή:

♦ ***Όταν παριστάνουμε τους αρνητικούς με συμπλήρωμα ως προς 2 κάνουμε πρόσθεση χωρίς να ενδιαφερόμαστε για τα πρόσημα των προσθεταίων.***

Να λοιπόν πώς δουλεύει ο υπολογιστής στην περίπτωση αυτή:

- οι αρνητικοί παριστάνονται με συμπλήρωμα ως προς 2,
- κατά την πρόσθεση το ψηφίο προσήμου δεν έχει διαφορετική μεταχείριση από τα άλλα,

- η πρόσθεση γίνεται με αριθμητική υπολοίπων ως προς 2^N, όπου N το πλήθος δυαδικών ψηφίων που χρησιμοποιούνται για την παράσταση.

Ποιος είναι ο μέγιστος αριθμός που μπορούμε να παραστήσουμε; Είναι ο

```
 7 6 5 4 3 2 1 0
|0|1|1|1|1|1|1|1| = 127₁₀
```

Η ελάχιστη τιμή είναι -128 και παριστάνεται ως:

```
 7 6 5 4 3 2 1 0
|1|0|0|0|0|0|0|0| = -128₁₀
```

Μάλλον θα θέλεις κάποια βοήθεια για να βρεις το "`-128`". Λοιπόν: αφού έχει "`1`" στο ψηφίο 7, είναι αρνητικός. Ας εφαρμόσουμε *αντιστρόφως* αυτά που κάναμε για να υπολογίσουμε το συμπλήρωμα ως προς 2:

Βήμα 1: αφαιρούμε το 1

```
  7 6 5 4 3 2 1 0
 |1|0|0|0|0|0|0|0|
-               1
 -----------------
 |0|1|1|1|1|1|1|1|
  7 6 5 4 3 2 1 0
```

Βήμα 2: αλλάζουμε τα 0 σε 1 και τα 1 σε 0:

`10000000`

που είναι το $2^7 = 128$.

Το μηδέν παριστάνεται με έναν μοναδικό τρόπο: "`00000000`".

Η C++ έχει τρεις τύπους που τους χειρίζεται με τον τρόπο αυτόν:

- `(signed) char`, σε 8 ψηφία. Παριστάνει τιμές από -128 μέχρι 127 με αριθμητική υπολοίπου ως προς $2^8 = 256$.
- `short int`, σε 16 ψηφία. Παριστάνει τιμές από -32 768 μέχρι 32 767 με αριθμητική υπολοίπου ως προς $2^{16} = 65\,536$.
- `int` και `long int`, σε 32 ψηφία. Παριστάνει τιμές από -2 147 483 648 μέχρι 2 147 483 647 με αριθμητική υπολοίπου ως προς $2^{32} = 4\,294\,967\,296$.

Να (ξανα)τονίσουμε ότι *τα παραπάνω μεγέθη δεν καθορίζονται από το πρότυπο* της *γλώσσας*. Ας πούμε ότι είναι συνηθισμένες τιμές.

Στη συνέχεια θα δούμε και μερικούς ακόμη ακέραιους τύπους.

Πού θα μας χρειαστούν όλα αυτά; Το συζητούμε στην συνέχεια.

17.2.1 * Ακέραιοι Τύποι του C99

Στο πρότυπο C99 (ISO/IEC 1999) της C εισάγονται ακέραιοι τύποι με 64 δυαδικά ψηφία: `long long int` και `unsigned long long int`.

- Ελάχιστη τιμή του `long long int`

  `LLONG_MIN`: `-9223372036854775807` = $-(2^{63} - 1)$
- Μέγιστη τιμή του `long long int`

  `LLONG_MAX`: `+9223372036854775807` = $2^{63} - 1$
- Μέγιστη τιμή του `unsigned long long int`

  `ULLONG_MAX`: `18446744073709551615` = $2^{64} - 1$

Οι `long long int` και `unsigned long long int` προβλέπονται και στο C++11.

Πέρα από αυτούς τους ακέραιους 64 δυαδικών ψηφίων, το C99 υποδεικνύει και μερικούς ορισμούς-μετονομασίες τύπων. Η πρώτη οικογένεια περιλαμβάνει ορισμούς ονομάτων της μορφής `int`*N*`_t` και `uint`*N*`_t`. Το *N* υποδηλώνει το πλήθος δυαδικών ψηφίων. Ο gcc (Dev C++), στο `stdint.h`, έχει τους εξής σχετικούς ορισμούς:

```
typedef signed char        int8_t;
typedef unsigned char      uint8_t;
typedef short              int16_t;
typedef unsigned short     uint16_t;
typedef int                int32_t;
```

```
typedef unsigned              uint32_t;
typedef long long             int64_t;
typedef unsigned long long    uint64_t;
```

Τι κερδίζουμε με αυτούς; Όταν θέλουμε μια ακέραιη μεταβλητή με 32 ψηφία δεν χρειάζεται να ψάχνουμε αν θα τη δηλώσουμε υποχρεωτικώς `long int` ή μας κάνει και ο `int`· δηλώνουμε:

```
   int32_t x;
```

και τελειώνουμε.

Μια άλλη οικογένεια περιλαμβάνει ονόματα της μορφής `int_least`*N*`_t` και `uint_least`*N*`_t` (υποχρεωτικό για *N*: 8, 16, 32, 64). Για παράδειγμα, ο τύπος `int_least32_t` είναι ο «μικρότερος« ακέραιος τυπος που έχει τουλάχιστον 32 ψηφία.

Μια τρίτη οικογένεια περιλαμβάνει ονόματα της μορφής `int_fast`*N*`_t` και `uint_fast`*N*`_t` (υποχρεωτικό για *N*: 8, 16, 32, 64). Ο τύπος `int_fast32_t` είναι ο «ταχύτερος» (ό,τι και αν σημαίνει αυτό) ακέραιος τυπος που έχει τουλάχιστον 32 ψηφία.

Τέλος, ο τύπος `intmax_t` είναι αυτός στον οποίον μπορούμε να παραστήσουμε οποιαδήποτε ακέραιη τιμή μπορεί να παρασταθεί στη συγκεκριμένη υλοποίηση. Ο αντίστοιχος για τιμές χωρίς πρόσημο είναι ο `uintmax_t`. O gcc, στο `stdint.h`, έχει:

```
typedef long long             intmax_t;
typedef unsigned long long    uintmax_t;
```

Όλοι αυτοί οι ορισμοί προβλέπεται να υπάρχουν στο νέο πρότυπο της C++.

17.3 Οι Ακέραιοι στο Πρόγραμμα

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
  int  i( INT_MAX );
  i = 4*i;
  cout << i << endl;
}
```

Αποτέλεσμα:

```
-4
```

Ας δούμε τί έγινε: Η (gcc - Dev) C++ παριστάνει τιμές τύπου `int` σε 32 δυαδικά ψηφία και το *INT_MAX* παριστάνεται ως εξής:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|0|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|
```

Το αποτέλεσμα του πολλαπλασιασμού υπολογίζεται σε:

```
  3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
  2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
  1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|0|0| = 8589934588
```

Αλλά, στον τύπο `int` δεν μπορούμε να παραστήσουμε αυτήν την τιμή· έχουμε **υπερχείλιση** (overflow). Δουλεύοντας με αριθμητική υπολοίπων και συμπλήρωμα ως προς 2, η συνέχεια είναι η εξής: στη μεταβλητή *i* αποθηκεύονται τελικώς τα ψηφία 0 μέχρι 31 του αποτελέσματος. Όταν έρθει η ώρα να εκτελεσθεί η εντολή `cout << i << endl`, πρώτα ελέγχεται το ψηφίο προσήμου (31)· αφού έχει τιμή 1, ο υπολογιστής καταλαβαίνει ότι πρόκειται για *αρνητική τιμή!* Στη συνέχεια, αφαιρεί 1:

```
 |1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|0|0|
-                                                                1
 -----------------------------------------------------------------
 |1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|0|1|1|
```

και αλλάζει το 1 σε 0 και τα 0 σε 1:

```
|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|
```

που είναι το 4.

Τι δίδαγμα βγαίνει από αυτά;

- ***Στις πράξεις του* `int` *(και των άλλων ακεραίων τύπων) είναι δυνατόν να έχουμε υπερχείλιση χωρίς καμιά ειδοποίηση από τον υπολογιστή. Αυτό είναι συχνά από τα δυσκολότερα λάθη, τουλάχιστον στην ανίχνευση.***

Πώς αντιμετωπίζεται αυτό το πρόβλημα; Ας δούμε τί μπορούμε να κάνουμε με την πρόσθεση. Έχουμε δυο μεταβλητές *x*, *y*, τύπου `int` και θέλουμε να υπολογίσουμε το $x + y$, αν υπολογίζεται. Για τα *x*, *y* έχουμε:

$$INT_MIN \le x \le INT_MAX$$
$$INT_MIN \le y \le INT_MAX$$

και θα αποπειραθούμε την πρόσθεση μόνον αν ξέρουμε από πριν ότι

$$INT_MIN \le x + y \le INT_MAX \text{ ή}$$
$$INT_MIN - y \le x \le INT_MAX - y$$

Φυσικά, δεν μπορούμε να γράψουμε:

```
if (INT_MIN-y <= x && (x <= INT_MAX-y)
  s = x + y;
else
  Λάθος
```

διότι οι πράξεις `INT_MIN - y` και `INT_MAX - y` *δεν είναι ασφαλείς!* Π.χ. αν η *y* έχει αρνητική τιμή, η `INT_MAX - y` μας δίνει σίγουρα υπερχείλιση. Ας τα ξαναδούμε πιο προσεκτικά. Κατ' αρχάς, αν οι *x*, *y* έχουν ετερόσημες τιμές δεν υπάρχει περίπτωση υπερχείλισης με την πρόσθεση. Αν οι *x*, *y* έχουν τιμές ≥ 0 τότε μας ενδιαφέρει μόνον ο έλεγχος $x \le INT_MAX - y$ και η πράξη δεξιά είναι ασφαλής. Αν οι *x*, *y* έχουν τιμές < 0 τότε μας ενδιαφέρει μόνον ο έλεγχος $INT_MIN - y \le x$ και η πράξη αριστερά είναι ασφαλής. Μπορούμε λοιπόν να γράψουμε την:

```
int addInt( int x, int y )
{
   int fv;

   if ( x >= 0 )
   {
      if ( y < 0 )  fv = x + y;
      else if ( x <= INT_MAX - y )  fv = x + y;
      else
         throw IntOvrflXptn( "addInt", 0, x, y );
   }
   else    // x < 0
   {
      if ( y >= 0 )  fv = x + y;
      else if ( INT_MIN - y <= x )  fv = x + y;
      else
         throw IntOvrflXptn( "addInt", 0, x, y );
   }
   return fv;
} // addInt
```

«Δηλαδή», σκέφτεσαι με φρίκη, «θα πρέπει αντί για “`c = a + b`” να γράφω “`c = addInt(a, b)`” και αντί για μια πράξη να καλώ μια συνάρτηση και να διαχειρίζομαι εξαιρέσεις;» Όχι! Σε ένα καλοσχεδιασμένο πρόγραμμα οι περισσότερες πράξεις *είναι* συνήθως ασφαλείς· και αυτές που δεν είναι φαίνονται εύκολα.[2] Φυσικά, δεν υπάρχει συνταγή για το πότε βάζουμε έλεγχο και πότε όχι· αλλά, δες δυο παραδείγματα:

```
#include <iostream>
#include <string>
#include <climits>
```

[2] Και ακόμη: για τις άλλες πράξεις τα πράγματα είναι πολύ πιο απλά.

```
using namespace std;

struct IntOvrflXptn
{
   char funcName[100];
   int  errCode;
   int  errVal1;
   int  errVal2;
   IntOvrflXptn( const char* fn, int ec, int ev1=0, int ev2= 0 )
   {
      strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal1 = ev1;  errVal2 = ev2;
   }
}; // IntOvrflXptn

int addInt( int x, int y );

int main()
{
   int x, y, z;

   cout << "Δώσε δύο θετικούς ακέραιους < 100" << endl;
   cin >> x >> y;
   try
   {
      z = addInt( x, y );
      cout << z << endl;
   // . . .
   }
   catch ( IntOvrflXptn& xp )
   {
      cout << xp.errVal1 << " + " << xp.errVal2
           <<"??? ΣΟΒΑΡΕΨΟΥ!" << endl;
   }
// . . .
   do
   {
      cout << "Δώσε δύο θετικούς ακέραιους < 100" << endl;
      cin >> x >> y;
   } while (x <= 0 || 100 < x || y <= 0 || 100 < y);
   z = x + y;
// . . .
} // main
```

Το μήνυμα που δίνεις πριν από την εντολή ανάγνωσης δεν σου εξασφαλίζει οτιδήποτε. Στην πρώτη περίπτωση δεν σε ενδιαφέρει να δεις αν ο χρήστης υπάκουσε στην οδηγία σου και προχωράς. Η χρήση της *addInt()* θα σε προφυλάξει από μια τιμή της *z* χωρίς νόημα. Στη δεύτερη περίπτωση, αφού δεν προχωράς παρά μόνο με «σωστές» τιμές των *x*, *y*, μπορείς να κάνεις την πρόσθεση χωρίς άλλον έλεγχο.

17.3.1 Παράμετροι “unsigned”

Στην §6.8 (και στην §13.3.1) δίναμε τον κανόνα:

- ***Μη βάζεις στις συναρτήσεις σου παραμέτρους τιμής τύπου*** `unsigned int`, `unsigned long int`, `unsigned short int`, `unsigned char` ***αλλά, αντιστοίχως:*** `int`, `long int`, `short int`, `char`. ***Μετά βάλε έλεγχο προϋπόθεσης.***

Στο παράδειγμα παραβίασης του κανόνα που δίναμε καλούσαμε

```
   n = -1024;
   cout << n << "  " << intSqrt(n) << endl;
```

τη συνάρτηση με επικεφαλίδα

```
unsigned int intSqrt( unsigned int x );
```

Και τι βλέπαμε; Στη συνάρτηση περνούσε στη *x* η τιμή 4294966272 και η συνάρτηση υπολόγιζε την (ακέραιη) τετραγωνική της ρίζα.

Τώρα μπορείς να καταλάβεις τι γίνεται. Σε `int` τεσσάρων ψηφιολέξεων το 1024 (= 2^{10}) παριστάνεται ως:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|0|0|0|0|0|0|0|
```

Το "`-1024`" –σε συμπλήρωμα ως προς 2– παριστάνεται ως:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|0|0|0|0|0|0|0|0|0|0|
```

Αυτή είναι η εσωτερική παράσταση της *n* και αντιγράφεται ως τιμή της *x*. Μέσα στη συνάρτηση αυτή η παράσταση ερμηνεύεται ως `unsigned int`. Έτσι προκύπτει η τιμή 4294966272.

Άσκηση για σένα: η επιβεβαίωση της παράστασης του "`-1024`" και ο υπολογισμός της τιμής 4294966272.

17.4 * Απαριθμητοί Τύποι (ξανά)

Πρωτοείδαμε τους απαριθμητούς τύπους πολύ νωρίς (§4.8) και τους χρησιμοποιούμε όπως θα χρησιμοποιούσαμε τους αντίστοιχους της Pascal –και πολύ καλά κάναμε. Στη συνέχεια, σε ορισμένες περιπτώσεις, θα πρέπει να τους χρησιμοποιήσουμε όπως τους χρησιμοποιεί η C.

Ας ξαναδούμε τον τύπο

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

και τη δήλωση:

```
    Digit d1( 49 );
```

Ο μεταγλωττιστής θα την απορρίψει: «invalid conversion from '`int`' to '*Digit*'» (gcc –Dev C++).[3] Αν όμως δώσεις:

```
    Digit d1( static_cast<Digit>(49) );
```

όλα πάνε μια χαρά. Σωστό!

Δοκιμάζουμε όμως και το εξής:

```
    d1 = static_cast<Digit>( 9 );
```

Και αυτό περνάει χωρίς το παραμικρό πρόβλημα, ενώ, με βάση αυτά που ξέρουμε, θα έπρεπε να γίνονται δεκτές τιμές από `static_cast<Digit>(49)` μέχρι και `static_cast <Digit>(57)`.

Αυτό που συμβαίνει στην πραγματικότητα είναι το εξής:

- Αν οι σταθερές που έχουμε στην απαρίθμηση είναι μη αρνητικές η C++ θα δεχτεί ως τιμή μιας σταθεράς κάθε ακέραιη τιμή από 0 μέχρι τη μέγιστη τιμή που μπορεί να παρασταθεί στα δυαδικά ψηφία –χωρίς να υπολογίζουμε ψηφίο προσήμου– που απαιτούνται για τη μέγιστη τιμή της απαρίθμησης. Στο παράδειγμά μας μέγιστη τιμή της απαρίθμησης είναι η *nine* που αντιστοιχεί στο $57_{10} = 111001_2$. Η μέγιστη τιμή που μπορεί να παρασταθεί σε έξη δυαδικά ψηφία είναι $111111_2 = 63_{10}$.
- Αν υπάρχουν και αρνητικές τιμές τότε ισχύουν τα ίδια αλλά θα πρέπει να υπολογίζουμε και ψηφίο προσήμου. Για παράδειγμα, αν στην απαρίθμηση του παραδείγματος βάλουμε άλλο ένα στοιχείο `neg = -1`, τότε η περιοχή είναι από -64 μέχρι 63 (παράσταση σε 7 ψηφία). Αν βάλουμε `neg = -100`, τότε η περιοχή είναι από -128 μέχρι 127 (παράσταση σε 8 ψηφία).

[3] Ο δικός σου μεταγλωττιστής, με ή χωρίς διαμαρτυρίες (warnings), τη δέχτηκε! Συμβαίνουν και αυτά...

7	6	5	4	3	2	1	0	
0	0	0	0	1	1	0	0	*x*
1	1	1	1	0	0	1	1	~*x*

Σχ. 17-1 Η ψηφιοπράξη "**~**" (συμπλήρωμα ως προς 1).

7	6	5	4	3	2	1	0	
0	0	0	0	1	1	0	0	*x*
0	0	1	0	0	1	0	0	*y*
0	0	0	0	0	1	0	0	*x* & *y*

Σχ. 17-2 Ψηφιοπράξη "**&**" (**and**).

Πάντως είναι πολύ πιθανό, ο μεταγλωττιστής σου να δεχτεί χωρίς διαμαρτυρίες ως τιμή μεταβλητής τύπου *Digit* τη `static_cast<Digit>(n)` όπου *n* τυχούσα τιμή τύπου `int`.

17.5 Ψηφιοπράξεις στη C++

Η C++ προσφέρει ορισμένες πράξεις που επιτρέπουν διαχείριση των τιμών όλων των ακέραιων τύπων της (δυαδικό) ψηφίο προς ψηφίο· για τον λόγο αυτόν τις ονομάζουμε **ψηφιοπράξεις** (bitwise operations).

Ας τις δούμε ξεκινώντας από τις πράξεις **ολίσθησης** (shift). Ας πούμε ότι έχουμε:

```
    unsigned char  x, y;
//  . . .
    x = 77;
```

Στη μνήμη θα αποθηκευτούν σε μια ψηφιολέξη τα εξής:

7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1

Αν δώσουμε την εντολή `y = x << 2` (ολίσθησε αριστερά κατά 2 δυαδικά ψηφία), θα συμβούν τα εξής:

- οι τιμές όλων των ψηφίων της *x* θα ολισθήσουν κατά 2 θέσεις προς τα αριστερά,
- τα 2 πρώτα (από αριστερά, 7 και 6) θα χαθούν,
- τα δυο τελευταία θα γίνουν 0.
- Ο τύπος του αποτελέσματος είναι ίδιος με τον τύπο του πρώτου ορίσματος.

Έτσι, στη θέση *y* θα αποθηκευτούν τα εξής:

7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0

Σχεδόν παρομοίως γίνεται και η ολίσθηση δεξιά. Αν δώσουμε την εντολή `y = x >> 2` (ολίσθησε δεξιά κατά 2 δυαδικά ψηφία), θα πάρουμε στη *y*:

7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1

Το «σχεδόν» τι αφορά; Το «γέμισμα» με μηδενικά:

- *Αν ο τύπος της x είναι* `unsigned` *τότε οι πρώτες θέσεις που «αδειάζουν» θα «γεμίσουν» με μηδενικά.*
- *Αν ο τύπος της x δεν είναι* `unsigned` *και το ψηφίο προσήμου είναι "1" το πώς θα γεμίσουν οι θέσεις που αδειάζουν μπορεί να αλλάζει από τον έναν μεταγλωττιστή στον άλλον.*

Ορίζονται και οι σχετικές συντομογραφίες για την εκχώρηση:

- Αντί για "`x = x << N`" μπορείς να γράφεις "`x <<= N`" και
- αντί για "`x = x >> N`" μπορείς να γράφεις "`x >>= N`".

Στα παραδείγματα αυτά με την ολίσθηση χάνονται "1". Για να δούμε τι συμβαίνει αν δεν έχουμε τέτοιες απώλειες. Ας πούμε ότι στο *x* έχουμε βάλει την τιμή 12, οπότε, όπως είπαμε, η εσωτερική παράσταση είναι: 00001100 (ψηφία 7 6 5 4 3 2 1 0).

Μετά τη "`y = x << 2`" η *y* γίνεται: 00110000 (ψηφία 7 6 5 4 3 2 1 0) .που είναι $2^5+2^4 = 48 = 12\times4$, ενώ μετά τη "`y = x >> 2`" η *y* γίνεται: 00000011 (ψηφία 7 6 5 4 3 2 1 0) .που είναι $2^1+2^0 = 3 = 12/4$,

Δηλαδή, μπορούμε να πούμε:

`x << N` σημαίνει $x \times 2^N$ και `x >> N` σημαίνει $x / 2^N$;

Ναι,

7	6	5	4	3	2	1	0	
0	0	0	0	1	1	0	0	*x*
0	0	1	0	0	1	0	0	*y*
0	0	1	0	1	1	0	0	*x* | *y*

Σχ. 17-3 Ψηφιοπράξη "|" (**or**).

7	6	5	4	3	2	1	0	
0	0	0	0	1	1	0	0	*x*
0	0	1	0	0	1	0	0	*y*
0	0	1	0	1	0	0	0	*x* ^ *y*

Σχ. 17-4 Ψηφιοπράξη "^" (**xor**).

- *αν δεν έχουμε υπερχείλιση* (για τη "`<<`") και
- *αν δεν έχουμε περιπλοκές από τα πρόσημα* (`char`, `short`, `int`, `long`).

Αυτές οι δυο πράξεις μας δίνουν τη δυνατότητα να γράφουμε από την C++ δυο εντολές του επεξεργαστή. Είναι λοιπόν πολύ ταχύτερες από τις αντίστοιχες αριθμητικές, αλλά *μην αρχίσεις να σκέφτεσαι να κάνεις έτσι πολλαπλασιασμούς και διαιρέσεις ακεραίων*: χρειάζονται προσοχή στη χρήση και οι πιθανότητες για λάθη είναι πάρα πολλές.

Παρατήρηση: ▶
Ο προσεκτικός αναγνώστης, διαβάζοντας για τους τελεστές ολίσθησης, "`<<`" και "`>>`", θα πρέπει να ανησύχησε: πώς δεν γίνεται μπέρδεμα με τους ίδιους (οπτικώς) τελεστές που χρησιμοποιούμε για γράψιμο και διάβασμα τιμών; Οι τελεστές ολίσθησης περιμένουν δύο ορίσματα που είναι ακέραιοι αριθμοί, ενώ οι τελεστές εισόδου/εξόδου χρειάζονται αριστερά κάποιο ρεύμα. Βεβαίως, σε ορισμένες περιπτώσεις χρειάζεται λίγη προσοχή. Αν γράψεις:

```
    int x = 12;
    cout << x << 2 << endl;
    cout << (x << 2) << endl;
```

θα πάρεις αποτέλεσμα:

```
122
48
```

Το 122 προέρχεται από την πρώτη εντολή εκτύπωσης, που λέει: γράψε την τιμή της *x*, που είναι 12 και στη συνέχεια γράψε και το 2· έτσι παίρνουμε αυτό το "122". Η δεύτερη εντολή λέει: τύπωσε το αποτέλεσμα της πράξης "`x << 2`", που, όπως είδαμε, είναι 48.◀

Η C++ μας επιτρέπει ακόμη τις πράξεις "~", "&", "|" και "^" μεταξύ τιμών ακέραιων τύπων (αντίστοιχες των "`!`", "&&", "`||`" και "`!=`" μεταξύ λογικών τιμών). Αν οι *x*, *y* είναι τύπου `unsigned char`:

- Η "~x" (Σχ. 17-1) προκύπτει από τη x με αλλαγή κάθε ψηφίου "`0`" σε "`1`" και κάθε ψηφίου "`1`" σε "`0`" (συμπλήρωμα ως προς 1).
- Η "x & y" προκύπτει από τις x και *y* με **and** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-2 βλέπεις ότι το μοναδικό "`1`" που προκύπτει είναι στο ψηφίο 2 διότι στη θέση αυτήν έχουν "`1`" και η x και η *y*.
- Η "x | y" προκύπτει από τις x και *y* με **or** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-3 βλέπεις ότι η x | *y* έχει "`1`" στις θέσεις
  - 2 που έχουν "`1`" και η x και η *y*,
  - 3 όπου έχει "`1`" η x και
  - 5 όπου έχει "`1`" η y.

  Αν δεν έχουμε πρόσημα και "`1`" στην ίδια θέση τότε το x | *y* είναι το ίδιο με το x + *y*. Π.χ. αν στη *y* είχαμε "`00100010`" (παράσταση του 34_{10}) τότε το x | *y* είναι "`00101110`" που είναι παράσταση του 46_{10} = 12 + 34.
- Η "x ^ y" προκύπτει από τις x και *y* με **xor** μεταξύ των αντίστοιχων ψηφίων τους. Στο Σχ. 17-4 βλέπεις ότι η x ^ *y* έχει "`1`" στις θέσεις
  - 3 όπου έχει "`1`" μόνον η x και
  - 5 όπου έχει "`1`" μόνον η y.

  Για τους τρεις διμελείς ορίζονται συντομογραφίες εκχώρησης:
- Αντί για "`x = x & y`" μπορείς να γράφεις "`x &= y`",

- αντί για “`x = x | y`” μπορείς να γράφεις “`x |= y`” και
- αντί για “`x = x ^ y`” μπορείς να γράφεις “`x ^= y`”.

Εδώ όμως χρειάζεται και πάλι προσοχή: Ας πούμε ότι έχεις:

```
    short int m( 3 );
    char c( -5 );
```

και θέλεις να υπολογίσεις το: `m |= c`. Η *c* έχει εσωτερική παράσταση: “`111110 11`”. Για να γίνει η ψηφιοπράξη “`|`” θα πρέπει η *m* και η *c* να έχουν εσωτερική παράσταση με το ίδιο πλήθος ψηφίων. Η *m* «προωθείται» σε `short int` με εσωτερική παράσταση

“`1111111111111011`” (= -5 σε `short int`)!!!

Μάλλον δεν είναι αυτό που θέλεις. Αν είχες δηλώσει:

```
    unsigned char c( 251 );
```

η *c* θα είχε την ίδια εσωτερική παράσταση (11111011) και με την προώθηση θα γίνονταν “`0000000011111011`”.

Μετά από όσα είπαμε μπορείς να καταλάβεις τη σύσταση του (CERT 2009):[4]

♦ ***Χρησιμοποίησε τους τελεστές ψηφιοπράξεων μόνο με ορίσματα τύπων*** `unsigned`.[5]

Ας δούμε τώρα ένα παράδειγμα χρήσης των “`<<`” και “`&`”.

Παράδειγμα

Θέλουμε να γράψουμε μια:

```
int bitValue( unsigned char b, int pos )
```

που θα μας επιστρέφει την τιμή του ψηφίου στη θέση *pos* της ψηφιολέξης *b*.

Πώς θα σκεφτούμε; Ας πούμε ότι έχουμε δηλώσει: `unsigned char t` και η *t* έχει “`0`” σε όλες τις θέσεις εκτός από την *pos* όπου έχει “`1`”. Πώς θα είναι τα ψηφία της `b & t`; Αφού η *t* έχει “`0`” σε όλες τις θέσεις εκτός από την *pos* και αφού ξέρουμε ότι *P* `&& false` ≡ `false`, το ίδιο θα ισχύει και για την `b & t`: θα έχει “`0`” σε όλες τις θέσεις εκτός από την *pos*. Στη θέση *pos*, όπου η *t* έχει “`1`”:

- Αν η *b* έχει “`0`” τότε και η `b & t` θα πάρει “`0`” και θα έχει τιμή 0 (μηδέν) αφού θα έχει παντού μηδενικά.
- Αν η *b* έχει “`1`” τότε και η `b & t` θα πάρει “`1`” και θα έχει τιμή ≠ 0 αφού θα έχει ένα μη μηδενικό ψηφίο.

Και πώς θα δώσουμε στην *t* την τιμή που θέλουμε; Έτσι:

```
   unsigned char t( 1 );
   t = t << pos;
```

δηλαδή:

- Δίνοντας στην *t* τιμή 1 βάζουμε 1 στο ψηφίο 0 της *t* και 0 σε όλα τα άλλα και
- με την “`t = t << pos`” μεταφέρουμε το 1, κατά *pos* θέσεις προς τα αριστερά, ενώ σε όλες τις άλλες θέσεις υπάρχουν 0.

Πριν γράψουμε τη συνάρτησή να παρατηρήσουμε ότι ενώ ορίζεται για όλες τις τιμές της *b*, δεν ορίζεται για τιμές της *pos* < 0 ή > 7· δηλαδή δεν είναι ολική.

Να λοιπόν πώς θα είναι η

```
int bitValue( unsigned char b, int pos )
{
   if ( pos < 0 || 7 < pos )   throw pos;

   unsigned char t( 1 );
   t <<= pos;
```

[4] Σύσταση *INT*13: “Use bitwise operators only on unsigned operands.”

[5] Αν μελετάς σωστά αυτό το κεφάλαιο, θα έχεις ήδη παραβιάσει τη συσταση και θα την παραβιάσεις αρκετές φορές ακόμη για να δεις εσωτερικές παραστάσεις αρνητικών αριθμών. Δεν πειράζει, αφού είναι για εκπαιδευτικούς λόγους!

```
   return ( ((b & t) != 0) ? 1 : 0 );
} // bitValue
```

Αν θέλεις να δεις την εσωτερική παράσταση του (`unsigned char`) "`12`" δηλώνεις

```
   unsigned char x( 12 );
```

ζητάς:

```
   for ( int pos(7); pos >= 0; --pos )
      cout << bitValue( x, pos );
   cout << endl;
```

και παίρνεις:

```
00001100
```

Με χρήση της *bitValue()* μπορούμε να γράψουμε τη:

```
void display( ostream& tout, unsigned char b )
{
   for ( int pos(7); pos >= 0; --pos )
      tout << bitValue( b, pos );
} // display
```

που σου είναι χρήσιμη αν μελετάς σοβαρά αυτό το κεφάλαιο.

Το παρακάτω πρόγραμμα μας δίνει τις εσωτερικές παραστάσεις των ακεραίων 12 και 20 χρησιμοποιώντας την *display()*:

```
#include <iostream>
using namespace std;

int bitValue( unsigned char b, int pos );
void display( ostream& tout, unsigned char b );

int main()
{
   unsigned char  x;
   int    k;

   try
   {
      x = 12;
      display( cout, x );  cout << endl;
      x = 20;
      display( cout, x );  cout << endl;
   }
   catch ( int& p )
   {
      cout << " η bitValue κλήθηκε με δεύτερο όρισμα "
           << p << endl;
   }
} // main
```

❦❦❦

17.6 Ψηφιοχάρτες και Συνηθισμένες Πράξεις

Παρ' όλο που στα παραδείγματά μας, μέχρι τώρα, το τελικό μας ενδιαφέρον φαίνεται να βρίσκεται στην τιμή της ψηφιολέξης ή, γενικώς, της συνολικής παράστασης, η διαχείριση δυαδικών ψηφίων είναι ενδιαφέρουσα καθ' εαυτή. Με τη διαχείριση δυαδικών ψηφίων, εκτός άλλων εφαρμογών, μπορούμε να υλοποιήσουμε σύνολα, στα οποία μας ενδιαφέρει μόνον η πληροφορία *ανήκει* ("1") ή *δεν ανήκει* ("0"). Σε τέτοιες περιπτώσεις χρησιμοποιούμε πίνακες ακέραιων τιμών, π.χ. τύπου `unsigned long`, που όμως τις βλέπουμε ως **ψηφιοσύνολα** (bitsets) ή **ψηφιοχάρτες** (bitmaps)[6].

[6] Ο όρος *bitmap* χρησιμοποιείται και σε έναν τρόπο παράστασης γραφικών (bitmap graphics).

Στη συνέχεια δίνουμε μερικές συναρτήσεις –για την ακρίβεια: περιγράμματα συναρτησεων– για μερικές πολύ συνηθισμένες περιπτώσεις διαχείρισης ψηφιοπινάκων. Μπορεί να κληθούν με τύπο πρώτης παραμέτρου (*T*) κάποιον από τους `int`, `unsigned int`, `long`, `unsigned long`, `short`, `unsigned short`, `char`, `unsigned char`, `wchar_t`, `long long`, `unsigned long long`.

Μερικές από αυτές ρίχνουν εξαίρεση τύπου:

```
struct BitmapXptn
{
   enum { outOfRange, paramErr };
   char funcName[100];
   int  errorCode;
   int  errVal1, errVal2;
   BitmapXptn( const char* mn, int ec, int v1, int v2 = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = v1;  errVal2 = v2;  }
}; // BitmapXptn
```

Προσοχή! Στα σχόλια τεκμηρίωσης (και μόνο) των περιγραμμάτων που δίνουμε στη συνέχεια θα χρησιμοποιούμε τον συμβολισμό `b[pos]` για το δυαδικό ψηφίο της *b* στη θέση *pos*. *Ο συμβολισμός αυτός δεν είναι δεκτός από τη C++ για τέτοια χρήση.*

17.6.1 Τιμή Δυαδικού Ψηφίου

Ξεκινούμε μετατρέποντας σε περίγραμμα τη *bitValue* που γράψαμε πιο πριν:

```
template < typename T >  int bitValue( T b, int pos )
```

Η βασική διαφορά είναι ότι τώρα το τελευταίο δυαδικό ψηφίο δεν βρίσκεται στη θέση 7, αλλά στη θέση

$$8(\texttt{sizeof}\ b) - 1$$

Έτσι έχουμε:

```
// bitValue -- επιστρέφει τη b[pos]
//             Προϋπόθεση: 0 <= pos < 8*(sizeof b)
template < typename T >
int bitValue( T b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) <= pos )
      throw BitmapXptn( "bitValue", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;
   return ( ((b & t) != 0) ? 1 : 0 );
} // bitValue
```

Μετατρέπουμε σε περίγραμμα και τη *display()* για να μπορείς να κάνεις τις δοκιμές σου:[7]

```
template < typename T >
void display( ostream& tout, T b )
{
   int lastb( 8*(sizeof b) - 1 );

   for ( int pos(lastb); pos >= 0; --pos )
      tout << bitValue( b, pos );
} // display
```

[7] Κανονικώς θα πρέπει να ελέγχουμε αν είναι ανοικτό το ρεύμα, αν το γράψιμο έγινε επιτυχώς και να ρίχνουμε τις αντίστοιχες εξαιρέσεις.

17.6.2 Βάλε Τιμή 1 σε Δυαδικό Ψηφίο

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  void setBit( T& b, int pos )
```

που θα αλλάζει το πρώτο όρισμα ώστε να έχει:

- "1" στο δυαδικό ψηφίο της θέσης *pos*.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις.

Φυσικά θα πρέπει να υπάρχει «*δυαδικό ψηφίο της θέσης pos*», δηλαδή:

$$0 <= pos < 8(\texttt{sizeof}\ b)$$

Πώς θα πετύχουμε το στόχο μας; Όπως μάθαμε, μετά τις

```
   T t( 1 );
   t <<= pos;
```

η *t* έχει "0" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "1". Η τιμή της παράστασης `b | t` θα έχει:

- "1" στο δυαδικό ψηφίο της θέσης *pos*, αφού η *t* έχει "1" και ξέρουμε ότι P `||` `true` ≡ `true`.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις, αφού η *t* έχει παντού "0" και ξέρουμε ότι P `||` `false` ≡ P.

Να λοιπόν το περίγραμμα της συνάρτησης:

```
// setBit -- αλλάζει την τιμή της πρώτης παραμέτρου βάζοντας
//           "1" στη θέση pos
//           Προϋπόθεση: 0 <= pos < 8*(sizeof b)
//           Απαιτηση:
//           βτελ[pos] == 1 && για κάθε k!=pos: βτελ[k]==βαρχ[k]
template < typename T >
void setBit( T& b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) - 1 < pos )
      throw BitmapXptn( "setBit", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;
   b |= t;
} // setBit
```

Ας δούμε δύο παραδείγματα χρήσης:

Παράδειγμα 1 ➾

Σε μια μεταβλητή *si* τύπου `unsigned short int` (16 δυαδικά ψηφία) θέλουμε να έχουμε όλα τα ψηφία "0" εκτός από αυτά που βρίσκονται στις θέσεις 10, 3 και 9:

```
      si = 0;
      setBit( si, 10);  setBit( si, 3);  setBit( si, 9);
      display( cout, si );  cout << endl;
```

Αποτέλεσμα:

```
0000011000001000
```

⬲⬲⬲

Παράδειγμα 2 ➾

Σε μια μεταβλητή *si* τύπου `unsigned short int` θέλουμε να εκχωρήσουμε την τιμή μιας άλλης μεταβλητής *uc* τύπου `unsigned char`. Θέλουμε ακόμη, το ψηφίο 13 της *si* να έχει τιμή "1":

```
      display( cout, uc );  cout << endl;
      si = uc;
      setBit( si, 13 );
      display( cout, si );  cout << endl;
```

Αποτέλεσμα:

```
11111011
0010000011111011
```

❦❦❦

Παρατήρηση: ▶
Ένα εύλογο ερώτημα που μπορεί να έχουν πολλοί είναι το εξής: Γιατί να μην ελέγξουμε αν το συγκεκριμένο ψηφίο είναι ήδη "1"; Στην περίπτωση αυτή δεν χρειάζεται να κάνουμε οτιδήποτε. Ας το δούμε· ο έλεγχος θα γίνει όπως είδαμε στη *bitValue()*:

```
   T t( 1 );
   t <<= pos;
   if ( (b & t) == 0 )  b |= t;
```

Δηλαδή:

- Σε κάθε περίπτωση έχουμε υπολογισμό της "`b & t`" και της **if** και
- Όταν το ψηφίο δεν έχει τιμή "1" εκτέλεση και της "`b |= t`".

Προφανώς ο τρόπος που επιλέξαμε –υπολογισμός μόνον της "`b |= t`"– είναι σαφώς πιο συμφέρων.◀

17.6.3 Βάλε Τιμή 0 σε Δυαδικό Ψηφίο

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  void clearBit( T& b, int pos )
```

που θα αλλάζει το πρώτο όρισμα ώστε να έχει:

- "`0`" στο δυαδικό ψηφίο της θέσης *pos*.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις.

Η λύση στο πρόβλημά μας μπορεί να προκύψει από τη λύση στο προηγούμενο πρόβλημα αν εναλλάξουμε τα "`0`" και "`1`" καθώς και τα **and** και **or**: Πράγματι, ας πούμε ότι η *t* έχει "`1`" σε όλες τις θέσεις εκτός από την *pos* όπου έχει "`0`". Η τιμή της παράστασης `b & t` θα έχει:

- "`0`" στο δυαδικό ψηφίο της θέσης *pos*, αφού η *t* έχει "`0`" και ξέρουμε ότι *P* `&& false` ≡ `false`.
- τις τιμές που έχει αρχικώς η *b* σε όλες τις άλλες θέσεις, αφού η *t* έχει παντού "`1`" και ξέρουμε ότι *P* `&& true` ≡ *P*.

Και πώς κάνουμε την t να «*έχει "`1`" σε όλες τις θέσεις εκτός από την pos όπου έχει "`0`"*»; Αυτό είναι απλό:

```
   T t( 1 );
   t <<= pos;  t = ~t;
```

Να λοιπόν η

```
// clearBit -- αλλάζει την τιμή της πρώτης παραμέτρου βάζοντας
//              "0" στη θέση pos
//              Προϋπόθεση: 0 <= pos < 8*(sizeof b)
//              Απαιτηση:
//              bτελ[pos] == 0 && για κάθε k!=pos: bτελ[k]==bαρχ[k]
template < typename T >
void clearBit( T& b, int pos )
{
   if ( pos < 0 || 8*(sizeof b) - 1 < pos )
      throw BitmapXptn( "clearBit", BitmapXptn::outOfRange, pos );

   T t( 1 );
   t <<= pos;  t = ~t;
   b &= t;
} // clearBit
```

Παράδειγμα ➢
Σε μια μεταβλητή *m* τύπου `unsigned short int` θέλουμε να εκχωρήσουμε την τιμή της *si* του Παραδ. 2, της προηγούμενης παραγράφου, με τη διαφορά ότι η *m* θα έχει “`0`” σε όλες τις άρτιες θέσεις:

```
      m = si;
      for ( int k(0); k < 8*sizeof(m); k += 2 )
         clearBit( m, k );
      display( cout, m );  cout << endl;
```

Αποτέλεσμα:
`0010000010101010`
⮈⮈⮈

17.6.4 Πλήθος "1"

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  size_t count1( T b )
```

που θα επιστρέφει ως τιμή το πλήθος των δυαδικών ψηφίων της που έχουν τιμή “`1`” σε τιμή *b* τύπου *T*.

Θα μπορούσαμε να καλέσουμε 8(`sizeof` *b*) φορές. Αντί για αυτό εξετάζουμε τόσες φορές την τιμή της `b & t` όπου η *t* –τύπου *T*– έχει μόνο ένα “`1`”. Κάθε φορά το “`1`” μετατοπίζεται ώστε να περάσει από όλες τις θέσεις της *t*.

```
// count1 -- επιστρέφει το πλήθος των ψηφίων της b με τιμή 1
//           Προϋπόθεση: true
//           Απαιτηση: fv == πληθος ψηφίων της b με τιμή 1
template < typename T >
size_t count1( T b )
{
   const size_t lastb( 8*(sizeof b) - 1 );
   T t( 1 );
   size_t fv( 0 );

   for ( int k(0); k <= lastb; ++k )
   {
      if ( (b & t) != 0 )  ++fv;
      t <<= 1;
   }
   return fv;
} // count1
```

Αν η *b* υλοποιεί κάποιο σύνολο (“`1`”: «ανήκει»), η *count*1 μας δίνει τον *πληθάριθμο* του συνόλου.

17.6.5 Μέρος Ψηφιοχάρτη

Θέλουμε ένα περίγραμμα συνάρτησης:

```
template < typename T >  T part( T b, int pos1, int pos2 )
```

που θα επιστρέφει ως τιμή τύπου *T* τα ψηφία της b από ***pos*1** μέχρι και ***pos*2**.

Να το δούμε με ένα παράδειγμα: Ας πούμε ότι έχουμε ψηφιοχάρτη 16 ψηφίων:

```
 1 1 1 1 1 1
 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|0|0|0|1|0|1|1|0|1|0|1|1|1|0|0|0|
             └─────────┘
```

Θέλουμε έναν ψηφιοχάρτη με το ίδιο μέγεθος που θα έχει το υπογραμμισμένο κομμάτι δηλαδή τα ψηφία από 4 μέχρι 9 και όλα τα άλλα ψηφία 0:

```
 1 1 1 1 1 1
 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|0|0|0|0|0|0|1|0|1|0|1|1|0|0|0|0|
             └─────────┘
```

Πρέπει δηλαδή να μηδενίσουμε τα ψηφία από 0 μέχρι 3 και από 10 μέχρι 15. Αυτό μπορεί να γίνει με την *clearBit*· μπορεί όμως να γίνει και ταχύτερα με τις πράξεις ολίσθησης. Αν στον αρχικό ψηφιοχάρτη κάνουμε μια ολίσθηση αριστερά και 6 (=15-9) θέσεις θα πάρουμε:

```
1 1 1 1 1 1
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
1 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0
```

Αν σε αυτό κάνουμε ολίσθηση δεξιά κατά 10 (= 15-9 + 4) θα πάρουμε:

```
1 1 1 1 1 1
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1
```

Τέλος, με ολίσθηση αριστερά κατά 4 θέσεις παίρνουμε αυτό που θέλουμε.

Αν πάρουμε υπόψη μας ότι: αντί για 15 έχουμε (γενικώς) το 8(`sizeof` *b*)–1, 4 είναι το *pos*1 και 9 είναι το *pos*2 έχουμε:

```
// part -- επιστρέφει το μέρος της b
//         με τα ψηφία της από pos1 μέχρι και pos2
//         Προϋπόθεση: 0 <= pos1 <= pos2 < 8*(sizeof v)
//         Απαίτηση: για κάθε k: 0..pos1-1 bτελ[k]==0 &&
//                   για κάθε k: pos1..pos2 bτελ[k]==bαρχ[k] &&
//                   για κάθε k: pos2+1..8*(sizeof v)-1 bτελ[k]==0
template < typename T >
T part( T b, int pos1, int pos2 )
{
   const size_t lastb( 8*(sizeof b) - 1 );

   if ( pos2 < pos1 )
      throw BitmapXptn( "part", BitmapXptn::paramErr, pos1, pos2 );
   if ( pos1 < 0 || pos2 < 0 || lastb < pos1 || lastb < pos2 )
      throw BitmapXptn( "part", BitmapXptn::outOfRange, pos1, pos2 );
// 0 <= pos1 <= pos2 <= lastb
   b <<= (lastb - pos2);
   b >>= (lastb - pos2 + pos1);
   b <<= pos1;
   return b;
} // part
```

Παράδειγμα ⮯

Αν η *si* είναι τύπου `unsigned short int` οι

```
      display( cout, si );  cout << endl;
      setBit( si, 12);
      unsigned short int sip( part(si, 4, 9) );
      display( cout, sip );  cout << endl;
```

δίνουν:

```
0000011010111000
0000001010110000
```

⮬⮬⮬

Και δύο λόγια για την πρώτη εξαίρεση: Για ορισμένες εφαρμογές το *pos*2 `<` *pos*1 δεν είναι και τόσο παράνομο. Απλώς στην περίπτωση αυτήν η συνάρτηση θα πρέπει να επιστρέφει *T*(0). Διαλέγεις και παίρνεις.

17.7 Τύποι *bitmask*

Πρωτοείδαμε τον τελεστή “`|`” στο Κεφ. 8, όταν συζητούσαμε για άνοιγμα ρεύματος. Ας πούμε ότι είχαμε να γράψουμε εμείς μια συνάρτηση που να ανοίγει ένα ρεύμα προς/από αρχείο. Γυρίζουμε λοιπόν στην §8.12 για να θυμηθούμε τα συστατικά του τρόπου ανοίγματος και καταλαβαίνουμε ότι (αν δεν θέλουμε να βάλουμε 6 ξεχωριστές παραμέτρους) θα πρέπει να βάλουμε μια παράμετρο τύπου:

```
struct OpenFlags
```

```
{
   bool app;
   bool ate;
   bool binary;
   bool in;
   bool out;
   bool trunc;
}; // OpenFlags
```

Επειδή μια τέτοια παράμετρος πιάνει 6 ψηφιολέξεις μπορούμε να κάνουμε οικονομία χρησιμοποιώντας ψηφιοπεδία:

```
struct OpenFlagsBF
{
   bool app:    1;
   bool ate:    1;
   bool binary: 1;
   bool in:     1;
   bool out:    1;
   bool trunc:  1;
}; // OpenFlagsBF
```

Μια τιμή τύπου *OpenFlagsBF* δεν χρειάζεται πάνω από μια ψηφιολέξη.

Σε αυτό το κεφάλαιο μάθαμε ότι μπορούμε να χειριστούμε το κάθε δυαδικό ψηφίο μιας τιμής ακέραιου τύπου. Και αφού οι τιμές που μπορεί να παίρνει είναι “`0`” ή “`1`” μπορούμε να το χειριστούμε ως τιμή τύπου `bool`. Έτσι, αντί για μεταβλητή τύπου *OpenFlagsBF* μας αρκεί μια μεταβλητή τύπου `unsigned char` (τη «μικρότερη» με 6 δυαδικά ψηφία). Πράγματι, ας πούμε ότι σε μια τέτοια τιμή ορίζουμε ότι το δυαδικό ψηφίο 0 ως σημαία για το “`app`”, το ψηφίο 1 για το “`ate`”, ..., το ψηφίο 5 για το “`trunc`”. Για να αποφύγουμε λάθη δηλώνουμε τις σταθερές:

```
const unsigned char appPos = 0, atePos = 1, binaryPos = 2,
                    inPos = 3, outPos = 4, truncPos = 5;
```

Έτσι, για να πούμε ότι θέλουμε να ανοίξουμε ένα ρεύμα *in*, *out* και *binary* σε μια μεταβλητή:

```
   unsigned char om;
```

βάζουμε:

```
   om = 0;
   setBit( om, inPos );  setBit( om, outPos );
   setBit( om, binaryPos );
```

Ελέγχουμε αν, ας πούμε, το δυαδικό ψηφίο στη θέση *binaryPos* έχει τιμή 1 ελέγχουμε αν `bitValue(om, binaryPos) == 1`.

Μπορούμε να τα καταφέρουμε χωρίς τη *setBit*; Ναι! Δες μια άλλη, διαφορετική, ομάδα σταθερών:

```
const unsigned char app = 1, ate = (app << 1),
                    binary = (app << 2), in = (app << 3),
                    out = (app << 4), trunc = (app << 5);
```

Αυτές δεν κρατούν τη θέση του ψηφίου που αντιστοιχεί στην κάθε σημαία *αλλά σε εκείνη ακριβώς τη θέση έχουν το μοναδικό* “`1`”.[8] Με αυτές δίνουμε στη *om* την τιμή που θέλουμε έτσι:

```
   om = in | out | binary;
```

Αφού οι σταθερές έχουν τα “`1`” σε διαφορετικές θέσεις θα μπορούσαμε να γράψουμε και:

```
   om = in + out + binary;
```

[8] Θα μπορούσαμε να είχαμε γράψει:

```
const unsigned char app = 1, ate = (1 << 1), binary = (1 << 2),
                    in = (1 << 3), out = (1 << 4), trunc = (1 << 5);
```

αλλά αυτό δεν είναι και πολύ καλή ιδέα!

Πάντως μπορούμε να κάνουμε και αυτό που κάνουμε στη *setBit*:

```
    om = 0;
    om |= in;  om |= out;  om |= binary;
```

Αν δεν έχουμε τη θέση πώς θα ελέγχουμε αν κάποιο ψηφίο έχει τιμή 1. Για να δούμε, ας πούμε, αν το δυαδικό ψηφίο στη θέση *binaryPos* έχει τιμή 1 ελέγχουμε αν `(om & binary) != 0`.

Τα παραπάνω είναι ένας τρόπος υλοποίησης ενός τύπου bitmask.

Ένας **τύπος bitmask** έχει τα εξής χαρακτηριστικά:

- N+1 σταθερές c_0 = `1`, c_1 = `(1 << 1)`, …, c_N`(1 << N)`.
- Αν v_1, v_2 τιμές του τύπου τότε και οι v_1 `&` v_2, v_1 `|` v_2, v_1 `^` v_2, `~`v_1 είναι τιμές του τύπου.
- Εκτός από τις ψηφιοπράξεις "`&`", "`|`", "`^`", "`~`" ορίζονται και οι «συντομογραφίες» εκχώρησης "`&=`", "`|=`", "`^=`".

Μπορούμε να υλοποιήσουμε έναν τέτοιον τύπο με τρεις τρόπους.

1. Ο πρώτος είναι *με κατάλληλο ακέραιο τύπο*, δηλαδή τύπο που οι τιμές του να παριστάνονται σε N+1 δυαδικά ψηφία τουλάχιστον. Παραπάνω, είδαμε παράδειγμα τέτοιας υλοποίησης. Αφού στην περίπτωσή μας N = 5 οποιοσδήποτε ακέραιος τύπος είναι κατάλληλος. Επιλέγουμε τον «μικρότερο»:

```
typedef unsigned char OpenMode;
const OpenMode app = 1, ate = (1 << 1), binary = (1 << 2),
               in = (1 << 3), out = (1 << 4), trunc = (1 << 5);
```

Όλες οι ψηφιοπράξεις που μας ενδιαφέρουν είναι ήδη ορισμένες.

2. Ο δεύτερος τρόπος υλοποίησης είναι με κάποιον *απαριθμητό τύπο*. Για το παράδειγμά μας ορίζουμε:

```
enum OpenMode
{  app = 1, ate = (1 << 1), binary = (1 << 2), in = (1 << 3),
   out = (1 << 4), trunc = (1 << 5)  };
```

Αν δεν διάβασες την §17.4 να πούμε ότι θα πρέπει να ξεχάσεις αυτά που μάθαμε για τους απαριθμητούς τύπους: σε μια μεταβλητή τύπου *OpenMode* μπορούμε (με ή χωρίς διαμαρτυρίες από τον μεταγλωττιστή) να βάλουμε τιμές που δεν υπάρχουν στην παραπάνω απαρίθμηση.

Αν και ορισμένοι μεταγλωττιστές θα δεχτούν –με διαμαρτυρίες (warnings)– να κάνουν ψηφιοπράξεις με αυτές τις σταθερές, το σωστό είναι να επιφορτώσεις τους τελεστές που είδαμε παραπάνω. Δίνουμε για το παράδειγμά μας την επιφόρτωση των "`&`" και "`&=`":

```
OpenMode operator&( OpenMode x, OpenMode y )
{
   return static_cast<OpenMode>( static_cast<unsigned char>(x) &
                                 static_cast<unsigned char>(y) );
} // operator&( OpenMode

OpenMode& operator&=( OpenMode& x , OpenMode y )
{
   x = x & y;
   return x;
} // operator&=( OpenMode
```

Όπως βλέπεις, ενώ με τη χρήση της απαρίθμησης δεν χρειάστηκε να επιλέξουμε ακέραιο τύπο, χρειάζεται τώρα για την επιφόρτωση των τελεστών.

3. Ο τρίτος τρόπος υλοποίησης είναι *με χρήση του περιγράμματος bitset* της STL. Θα τον δούμε αργότερα.

Οι τύποι bitmask χρησιμοποιούνται πολύ συχνά σε προγράμματα και βιβλιοθήκες C++ (και C).

17.8 Αριθμητικές Πράξεις και Ψηφιοπράξεις

Σε υποσημείωση της §8.6 λέγαμε για την "`ios_base::in|ios_base::out`" «*αυτό μπορεί να το δεις και ως:* "`ios_base::in+ios_base::out`".» Στην προηγούμενη παράγραφο είδαμε γιατί μπορεί να γραφεί κάτι τέτοιο, τουλάχιστον για τον πρώτο τρόπο υλοποίησης ενός τύπου *bitmask*. Αλλά,

- για τον τρίτο τρόπο υλοποίησης, η πρόσθεση δεν είναι δεκτή ενώ
- για τον δεύτερο τρόπο θα πρέπει να βάλεις τυποθεωρήσεις ώστε να περνάει από όλους τους μεταγλωττιστές.

Φυσικά, μπορεί να το δεις γραμμένο κάπου αλλού αλλά εσύ *δεν θα πρέπει να το γράφεις* με βάση το εξής σκεπτικό: Αφού αυτό που κάνουμε είναι διαχείριση δυαδικών ψηφίων και όχι αριθμητική πράξη γιατί να βάλουμε το "+"; Μόνο και μόνο επειδή δουλεύει;

Στις προηγούμενες παραγράφους επισημάναμε ότι είναι επικίνδυνο και το αντίστροφο: να προσπαθήσεις να πάρεις αριθμητικά αποτελέσματα με ψηφιοπράξεις. Τα αριθμητικά αποτελέσματα θα τα παίρνεις με αριθμητικές πράξεις.

Καταλήγουμε λοιπόν στη σύσταση του (CERT 2009):[9]

♦ ***Απόφυγε να κάνεις ψηφιοπράξεις και αριθμητικές πράξεις στα ίδια δεδομένα.***

17.9 Παράσταση και Πράξεις στον Τύπο "`float`"

Όπως ο `int` σε σχέση με το σύνολο $\mathbb{Z}$ των ακεραίων, έτσι και ο τύπος `float`, σε σύγκριση με το σύνολο $\mathbb{R}$ των πραγματικών, έχει δυο σοβαρούς περιορισμούς:

- Υπάρχει μέγιστος και ελάχιστος αριθμός τύπου `float`.
- Κάθε αριθμός τύπου `float` παριστάνεται με πεπερασμένο πλήθος ψηφίων.

Συνήθως, έναν πραγματικό αριθμό x τον γράφουμε ως:

$\sigma\ \psi_n\, \psi_{n-1} \ldots \psi_0 \,.\, \psi_{-1}\, \psi_{-2} \ldots$

και με αυτό εννοούμε ότι:

$$x = \sigma(\psi_n 10^n + \psi_{n-1} 10^{n-1} \ldots + \psi_0 10^0 + \psi_{-1} 10^{-1} + \psi_{-2} 10^{-2} \ldots)$$

όπου σ: πρόσημο (+ ή –) και ψ_k: δεκαδικό ψηφίο.

Φυσικά, ο ΗΥ δεν μπορεί να αποθηκεύσει στην μνήμη του τα άπειρα ψηφία ενός άρρητου αριθμού. Κάθε αριθμός παριστάνεται με πεπερασμένο αριθμό ψηφίων. Έχουμε λοιπόν **απώλεια σημαντικών ψηφίων** (loss of significant digits). Αυτό το σφάλμα, που εισάγεται με την παράσταση των πραγματικών αριθμών, μεγαλώνει με την εκτέλεση των πράξεων μεταξύ τους.

Ας υποθέσουμε ότι δουλεύουμε σε έναν **δεκαδικό** υπολογιστή. Κάθε τιμή τύπου `float`, για να αποθηκευτεί, μετατρέπεται στη μορφή:

$$M \times 10^e \qquad (1)$$

και αποθηκεύονται η **μαντίσα** (mantissa) M και ο **εκθέτης** (exponent) e. Η μαντίσα έχει τη μορφή:

$$\sigma\ 0 \,.\, \psi_1\, \psi_2 \psi_3 \psi_4 \psi_5$$

όπου σ το πρόσημο και ψ_k, $k = 1..5$ τα πέντε πιο σημαντικά ψηφία του αριθμού (με στρογγύλευση). Ο εκθέτης είναι διψήφιος ακέραιος και έχει τη μορφή:

$$\sigma\ \varepsilon_1\, \varepsilon_2$$

Η παράσταση στη μορφή (1) γίνεται μονοσήμαντη αν κάνουμε τη σύμβαση ότι *το* ψ_1 *δεν μπορεί να είναι μηδέν*. Στην περίπτωση αυτήν λέμε ότι η παράσταση είναι **κανονικοποιημένη** (normalized). Θα παριστάνουμε την αποθηκευμένη πληροφορία, στη μορφή:

$$\sigma\psi_1\, \psi_2\psi_3\psi_4\psi_5 : \sigma\varepsilon_1\, \varepsilon_2$$

---

[9] Σύσταση *INT*14: "*Avoid performing bitwise and arithmetic operations on the same data.*"

Αυτό είναι ένα παράδειγμα αποθήκευσης σε μορφή **κινητής υποδιαστολής** (floating point).

Το γνωστό μας π = 3.1415926535..., θα αποθηκευτεί ως:

+31416 : +01

(Θα γράφουμε π_f = +31416 : +01 ή π_f = 3.1416).

Η μέγιστη θετική τιμή που μπορεί να αποθηκευτεί είναι η:

+99999 : +99 (= 0.99999×10^{99})

και η ελάχιστη θετική τιμή:

+10000 : -99 (= 0.1×10^{-99})

Ένα χαρακτηριστικό της υλοποίησης του τύπου `float` είναι ο ελάχιστος θετικός ε που αν προστεθεί στο 1 μας δίνει τιμή μεγαλύτερη από 1:

$$\varepsilon = \min_{u>0}\{u | (1+u)_f \neq 1_f\}$$

Στον υπολογιστή μας, το 1 παριστάνεται ως:

+10000 : +01

και προφανώς η ελάχιστη αλλαγή που μπορούμε να του κάνουμε, είναι στο τελευταίο σημαντικό ψηφίο, κατά 1:

+10001 : +01 (= $0.10001 \times 10^1 = 1.0001$)

Έχουμε λοιπόν ότι: ε = 0.0001 .

Προσοχή όμως! Αυτό δεν σημαίνει ότι για κάθε $x \in$ `float` θα έχουμε και $(x + \varepsilon)_f \neq X_f$. Για παράδειγμα: $(10 + \varepsilon)_f = 10_f$ (Άσκ. 14-1). Πάντως το ε μας δείχνει πόσα σημαντικά ψηφία μπορούμε να παραστήσουμε: εφ' όσον μπορούμε να παραστήσουμε το 1+ε = 1.0001 μπορούμε να παραστήσουμε 5 σημαντικά ψηφία. Αν αυτό φαίνεται τετριμμένο τώρα που δουλεύουμε στο δεκαδικό σύστημα, δεν είναι τετριμμένο όταν δουλεύουμε στο δυαδικό ή στα παράγωγά του, που δεν μας είναι και τόσο οικεία.

Μπορούμε λοιπόν να πούμε ότι ο τύπος `float` είναι υποσύνολο του συνόλου των πραγματικών:

- `float` $\subset \mathbb{R}$ *(ακριβέστερα:* `float` $\subset \mathbb{Q}$ *(ρητοί))*

Η αποθήκευση πραγματικών τιμών στον υπολογιστή, είναι μια απεικόνιση από το σύνολο $\mathbb{R}$ στο υποσύνολό του `float`. Μερικές ιδιότητες αυτής της απεικόνισης θα δούμε στη συνέχεια.

Όπως φαίνεται από το παράδειγμά μας με το π, η αποθήκευση δεν γίνεται με ακρίβεια. Πάντως, για κάθε υπολογιστή, υπάρχει η εγγύηση ότι δυο τουλάχιστον πραγματικοί αριθμοί παριστάνονται με ακρίβεια: το 0 (μηδέν) και το 1 (ένα)!

Το 0 παριστάνεται ως:

+00000 : +00

και το 1 ως:

+10000 : +01 (= $0.1 \times 10^1 = 1$)

- **0_f == 0 *και* 1_f == 1**

Ο δείκτης $_f$ υποδηλώνει την παράσταση στο σύνολο `float`.

Αν μια πραγματική τιμή α παριστάνεται στο σύνολο `float` με την α_f, τότε και η -α παριστάνεται στο `float` και μάλιστα με την $-\alpha_f$:

$(-\alpha)_f$ == $-\alpha_f$

Όπως είδαμε παραπάνω, η τιμή 3.1415926535..., θα αποθηκευτεί ως:

+31416 : +01

Αλλά με τον ίδιο τρόπο θα αποθηκευτεί και η τιμή 3.14156 και η 3.1416 κλπ. Γενικά:

- ***Αν οι τιμές α_1, α_2 παριστάνονται στον υπολογιστή με την ίδια τιμή $\tau \in$ `float`, τότε με την ίδια τιμή παριστάνονται όλες οι τιμές του διαστήματος $[\alpha_1, \alpha_2]$.***

Από την ιδιότητα αυτή βγαίνουν τα εξής:

αν $\alpha > \beta$ τότε $\alpha_f \geq \beta_f$

αν $\alpha == \beta$ τότε $\alpha_f == \beta_f$

αν $\alpha < \beta$ τότε $\alpha_f \leq \beta_f$

Ας δούμε τώρα τι γίνεται με τις πράξεις.

Το πρώτο που θα δούμε είναι η **υπερχείλιση** και η **υποχείλιση**: είναι δυνατόν το αποτέλεσμα μιας πράξης να μην παριστάνεται γιατί είναι πολύ μεγάλο ή πολύ μικρό. Οι τιμές:

+60000 : +99 ($= 0.60000 \times 10^{99}$)

και

+70000 : +99 ($= 0.70000 \times 10^{99}$)

παριστάνονται στον υπολογιστή μας χωρίς πρόβλημα. Το άθροισμά τους όμως, $1.30000 \times 10^{99} = 0.13000 \times 10^{100}$ *δεν μπορεί να παρασταθεί* διότι είναι πολύ μεγάλο για τον υπολογιστή μας! Έχουμε δηλαδή **υπερχείλιση** (overflow). Τι θα κάνει ο υπολογιστής σε μια τέτοια περίπτωση; Σίγουρα θα σου δώσει μήνυμα για την κατάσταση που δημιουργήθηκε· στις περισσότερες περιπτώσεις θα σταματήσει και την εκτέλεση του προγράμματος.

Όπως καταλαβαίνεις, τέτοια αποτελέσματα μπορεί να βγουν και από τις τέσσερις πράξεις της αριθμητικής, όταν τις εκτελεί ο υπολογιστής. Δηλαδή:

♦ ***Το σύνολο*** **`float`** ***δεν είναι κλειστό ως προς τις πράξεις +, –, * και / .***

Καταλαβαίνεις ακόμη, ότι η υπερχείλιση στον τύπο `float` είναι λιγότερο επικίνδυνη από αυτήν του `int`, αφού αποφεύγεις την περίπτωση να συνεχιστεί η εκτέλεση του προγράμματος με τιμές που δεν έχουν νόημα. Έχει νόημα να γράψουμε κάτι σαν *addFloat*, για ασφαλή πρόσθεση τιμών `float`; Ναι, διότι συχνά ξέρεις τί πρέπει να κάνεις σε περίπτωση υπερχείλισης και οπωσδήποτε είναι ενοχλητικό να βλέπεις το πρόγραμμα να σταματάει, έστω και αν σου γνωστοποιεί τί έγινε.

Αν από τον:

+11000 : -99 ($= 0.11 \times 10^{-99}$)

αφαιρέσουμε τον:

+10000 : -99 ($= 0.1 \times 10^{-99}$)

το αποτέλεσμα $0.01 \times 10^{-99} = 0.1 \times 10^{-100}$ δεν μπορεί να παρασταθεί στον υπολογιστή μας, γιατί είναι πολύ μικρό! Στην περίπτωση αυτή λέμε ότι έχουμε **υποχείλιση** (underflow). Συνήθως, ο υπολογιστής σε μια τέτοια περίπτωση θα βάλει το αποτέλεσμα 0 (μηδέν) χωρίς ειδοποίηση για το τι έγινε. Αλλά, αυτό δεν είναι και τόσο τραγικό!

Ας δούμε τώρα ένα άλλο επακόλουθο της πεπερασμένης παράστασης. Έστω ότι θέλουμε να προσθέσουμε με τον υπολογιστή μας τους αριθμούς 14.563 και 0.16773. Η αποθήκευσή τους θα γίνει με ακρίβεια:

+14563 : +02 ($= 0.14563 \times 10^{2} = 14.563$)

και

+16773 : +00 ($= 0.16773 \times 10^{0} = 0.16773$)

Για να τους προσθέσει ο υπολογιστής θα πρέπει πρώτα να τους μετασχηματίσει ώστε να έχουν τον ίδιο εκθέτη. Για την ακρίβεια μετασχηματίζει την τιμή με το μικρότερο εκθέτη (στρογγυλεύοντας σε πέντε θέσεις)[10]:

---

[10] Συνήθως, η Αριθμητική Μονάδα του υπολογιστή θα κάνει τις πράξεις με μεγαλύτερη ακρίβεια, αλλά η τιμή που θα μας επιστρέψει τελικά είναι σύμφωνη με τη μορφή που έχουμε στην αποθήκευση. Στην περίπτωσή μας, η δεύτερη τιμή θα γίνει:

+0016773 : +02

Στη συνέχεια θα γίνει η πρόσθεση:

+1473073 : +02

Αλλά, το αποτέλεσμα που θα είναι διαθέσιμο στο πρόγραμμά μας θα είναι:

+14731 : +02

+00168 : +02

Στη συνέχεια κάνει την πρόσθεση:

+14731 : +02

Βλέπουμε λοιπόν, ότι και οι πράξεις εισάγουν **σφάλματα στρογγύλευσης** (roundoff errors). Πρόσεξε ότι, αν προσπαθούσαμε να προσθέσουμε στο 14.563 τον 0.0001, το αποτέλεσμα θα ήταν 14.563!

Θα πρέπει να έχει γίνει πια φανερό, ότι όπως ξεχωρίσαμε τις ακέραιες τιμές από τις παραστάσεις τους στον τύπο `int`, *θα πρέπει να ξεχωρίσουμε και τις πράξεις των πραγματικών από αυτές του υπολογιστή για τον τύπο* `float`. Θα παριστάνουμε λοιπόν με:

$+_f \quad -_f \quad *_f \quad /_f$

τις πράξεις:

$+ \quad - \quad \times \quad /$

όπως εκτελούνται στον υπολογιστή.

Στη συνέχεια θα δούμε μερικές «αναμενόμενες» ιδιότητες των πράξεων αλλά και μερικές «περίεργες». Το σύμβολο της ισότητας θα έχει πιο γενικό νόημα από ότι συνήθως: π_1 == π_2, όπου π_1, π_2 αριθμητικές παραστάσεις, θα σημαίνει ότι: αν μεν οι πράξεις γίνουν χωρίς πρόβλημα (υπερχείλιση ή υποχείλιση) οι τιμές των δύο παραστάσεων θα είναι ίσες· θα έχουμε υπερχείλιση ή υποχείλιση αριστερά αν και μόνον αν έχουμε το ίδιο πράγμα και δεξιά.

Η αντιμεταθετικότητα της πρόσθεσης και του πολλαπλασιασμού ισχύει στον τύπο `float`:

- ***Αν a, b*** $\in$ `float` ***τότε*** $a +_f b = b +_f a$ ***και*** $a *_f b = b *_f a$

Αυτές οι ισότητες ισχύουν με το γενικευμένο νόημα που δώσαμε πιο πάνω.

Η προσεταιριστικότητα της πρόσθεσης δεν ισχύει! Ας δούμε ένα

Παράδειγμα ➾

Οι αριθμοί $a = 0.10000\times10^{94}$, $b = 0.55000\times10^{99}$, $c = -0.50000\times10^{99}$ παριστάνονται στον υπολογιστή μας με ακρίβεια. Το άθροισμα:

$$(a +_f b) +_f c = (0.000001 +_f 0.55000)\times10^{99} +_f (-0.50000\times10^{99})$$
$$= 0.55000\times10^{99} +_f (-0.50000\times10^{99})$$
$$= 0.50000\times10^{98}$$

δεν είναι ίσο με το:

$$a +_f (b +_f c) = 0.10000\times10^{94} +_f (0.55000\times10^{99} +_f 0.50000\times10^{99})$$
$$= 0.10000\times10^{94} +_f 0.05000\times10^{99}$$
$$= 0.50001\times10^{98}$$

⇜⇜⇜

Παρ' όλα αυτά, αν: $a, b \in$ `float` και $a \geq b \geq 0$, ισχύει η γνωστή μας ιδιότητα:

$(a -_f b) +_f b$ == a

δηλαδή: η πρόσθεση ακυρώνει την αφαίρεση.

Προβλήματα έχουμε και με την προσεταιριστικότητα του πολλαπλασιασμού: Αν πάρουμε: $a = 0.10000\times10^{60}$, $b = 0.10000\times10^{60}$, $c = 0.10000\times10^{-60}$ τότε το γινόμενο:

$$(a *_f b) *_f c = (0.10000\times10^{60} *_f 0.10000\times10^{60}) *_f 0.10000\times10^{-60}$$
$$= 0.10000\times10^{119} *_f 0.10000\times10^{-60} \text{ (υπερχείλιση!!)}$$

δεν είναι ίσο με το:

$$a *_f (b *_f c) = 0.10000\times10^{60} *_f (0.10000\times10^{60} *_f 0.10000\times10^{-60})$$
$$= 0.10000\times10^{60} *_f 0.10000\times10^{-1}$$
$$= 0.10000\times10^{58}$$

Οι παραπάνω ενδεικτικές επισημάνσεις δεν εξαντλούν πλήρως τα προβλήματα του τύπου **float**, που ξεκινούν από την πεπερασμένη παράσταση. Αλλά δείχνουν μερικά χαρακτηριστικά σημεία που πρέπει να προσέχεις όταν γράφεις αριθμητικά προγράμματα.

17.9.1 Υπολογισμός Περιοδικής Συνάρτησης

Ας ξαναδούμε τώρα κάτι που μάθαμε παλιά υπό το φως αυτών που είδαμε παραπάνω.

Στην §7.8 μάθαμε να κάνουμε αναγωγή της τιμής του ορίσματος μιας περιοδικής συνάρτησης στο διάστημα ορισμού με τις εντολές:

```
    x0 = x;
    while ( x0 >= b )  x0 = x0 - T;
 // (x0 < b) && (f(x0) == f(x))
    while ( x0 < a )   x0 = x0 + T;
 // (a ≤ x0 < b) && (f(x0) == f(x))
```

Τι θα γίνει αν η απόλυτη τιμή του x είναι πολύ μεγαλύτερη από την τιμή της περιόδου T; Στην περίπτωση αυτήν η εντολή `x0 = x0 - T` (ή `x0 = x0 + T`) *δεν αλλάζει την τιμή της* $x0$ και *η εκτέλεση της αντίστοιχης **while** δεν τελειώνει*. Θα γράφαμε πιο σωστά:

```
    x0 = x;
    if (x0 >= b)
    {
       if (x0 - T == x0)
       {
          throw "η τιμή της f δεν υπολογίζεται";
       }
       else
       {
          while (x0 >= b)  x0 = x0 - T;
       // (x0 < b) && (f(x0) == f(x))
       }
    }
    else if (x0 < a)
    {
       if (x0 + T == x0)
       {
          throw "η τιμή της f δεν υπολογίζεται";
       }
       else
       {
          while (x0 < a)   x0 = x0 + T;
       // (a ≤ x0 < b) && (f(x0) == f(x))
       }
    }
```

Θα πεις: καλύτερα τότε να δουλεύουμε με τον γρήγορο τρόπο αναγωγής. Ναι, αλλά πρόσεξε: είναι καλύτερο να γράψουμε:

```
    x0 = x - T*floor((x-a)/T);
 // (a ≤ x0 < b) && (f(x0) == f(x))
```

Όμως και στην περίπτωση αυτή, αν το $x - a$ είναι πολύ μεγαλύτερο από την περίοδο T τότε η $x0$ θα πάρει τιμή που δεν έχει και πολύ νόημα, πιθανότατα 0. Ακόμη, αν η περίοδος είναι μικρότερη από 1 είναι δυνατόν να έχουμε υπερχείλιση στη διαίρεση $(x - a)/T$.

17.10 Άλλοι Τύποι Κινητής Υποδιαστολής

Για να αντιμετωπισθούν τα προβλήματα του τύπου **float** η C++ –και άλλες γλώσσες προγραμματισμού– δίνουν στον προγραμματιστή άλλους αριθμητικούς τύπους με μερικές βελτιώσεις.

Η πιο συνηθισμένη περίπτωση είναι ο «διπλός `float`», που η C++ ονομάζει `double`. Οι τιμές αυτού του τύπου αποθηκεύονται σε χώρο (μνήμης) *διπλό* απ' όσον πιάνουν οι τιμές του τύπου `float`. Η αποθήκευση γίνεται συνήθως με κάποιον από τους παρακάτω τρόπους:

- Ο αριθμός θεωρείται σαν άθροισμα δύο τιμών τύπου `float`.
  - Η πρώτη έχει τα περισσότερο σημαντικά ψηφία του αριθμού και
  - η δεύτερη τα λιγότερο σημαντικά ψηφία.

  Για παράδειγμα, η τιμή 12.34567891234 θα γραφεί ως:

  12.345678912 = 12.345 + 0.00067891
  = $0.12345\times10^2 + 0.67891\times10^{-3}$

  και στις δυο θέσεις θα αποθηκευτούν τα:

  +12345:+02 και +67891:-03

- Ο δεύτερος τρόπος διαφέρει ως προς το περιεχόμενο της δεύτερης θέσης: εκεί απόθηκεύονται μόνο (τα λιγότερο) σημαντικά ψηφία. Δηλαδή, η δεύτερη θέση δεν είναι οργανωμένη όπως οι θέσεις τύπου `float` (δεν αποθηκεύεται εκθέτης). Για το παραπάνω παράδειγμα θα αποθηκευτούν τα εξής:

  +12345:+02 και 6789123

- Μια παραλλαγή του δεύτερου τρόπου, που επικρατεί στους νέους υπολογιστές, επιτρέπει και την αύξηση των θέσεων του εκθέτη. Στην περίπτωση αυτήν η περιοχή τιμών που μπορεί να παρασταθεί είναι ευρύτερη από αυτήν που παριστάνεται στον τύπο `float`. Σε μια τέτοια περίπτωση ο αριθμός μας θα αποθηκεύονταν ως:

  +1234:+002 και 5678912

  Ανακεφαλαιώνοντας, μπορούμε να πούμε για τον "διπλό `float`":

- Επιτρέπει την αποθήκευση περίπου διπλού πλήθους σημαντικών ψηφίων απ' ότι ο τύπος `float`.
- Μπορεί να επιτρέπει την παράσταση ευρύτερης περιοχής τιμών απ' ότι ο τύπος `float`, αλλά αυτό δεν είναι αναγκαίο.
- Κάθε τιμή του καταλαμβάνει διπλό χώρο μνήμης απ' ότι ο τύπος `float`.

  Φυσικά, τα προβλήματα του τύπου `float` δεν λύνονται· απλώς μετατοπίζονται.

17.11 Ο Τύπος "`float`" στο Δυαδικό Σύστημα

Μέχρι τώρα χρησιμοποιήσαμε το δεκαδικό σύστημα για να δούμε μερικά από αυτά που συμβαίνουν στους τύπους `float`. Αλλά οι ψηφιακοί υπολογιστές δουλεύουν συνήθως με το **δυαδικό** (binary) σύστημα και τα παράγωγά του **οκταδικό** (octal) και **δεκαεξαδικό** (hexadecimal). Τώρα θα δούμε μερικά από τα παραπάνω στο δυαδικό σύστημα.

Στο δυαδικό σύστημα η παράσταση μιας τιμής τύπου `float` γίνεται ως εξής:

$$M \times 2^e$$

Η **μαντίσα** M και ο **εκθέτης** e που αποθηκεύονται, είναι δυαδικοί αριθμοί. Ο εκθέτης είναι ακέραιος και έχει τη μορφή:

$s\ \varepsilon_1\ \varepsilon_2 \ldots\ \varepsilon_L$

όπου s πρόσημο και ε_k, $k = 1\,..\,L$ τα L δυαδικά ψηφία.

Η μαντίσα μπορεί να έχει τη μορφή:

$$s\ 0\,.\,\psi_1\,\psi_2 \ldots \psi_N$$

ή

$$s\ \psi_1\,\psi_2 \ldots \psi_N\,.\,0$$

Συνήθως η παράσταση κανονικοποιείται, δηλαδή *το ψ_1 δεν μπορεί να είναι μηδέν*. Στο δυαδικό σύστημα αυτό σημαίνει ότι $\psi_1 = 1$.

Σχ. 17-5 Παράσταση κινητής υποδιαστολής.

Μια τέτοια παράσταση βλέπουμε στο Σχ. 17-5.

Στο ψηφίο 31 αποθηκεύεται το πρόσημο του αριθμού –"`0`" για το "`+`", "`1`" για το "`-`". Στα ψηφία 30 - 23 αποθηκεύεται ο εκθέτης: Στο ψηφίο 30 το πρόσημό του (όπως παραπάνω) και στα ψηφία 29 - 23 η απόλυτη τιμή του. Στα ψηφία 22 - 0 αποθηκεύεται η μαντίσα. Το πρώτο της ψηφίο (22) είναι πάντοτε 1, εκτός αν ο αριθμός είναι 0 (μηδέν). Η υποδιαστολή νοείται ακριβώς πριν από το ψηφίο αυτό (22). Δηλαδή:

$$e=\begin{cases}\varepsilon_{29}\varepsilon_{28}...\varepsilon_{23}, & \alpha\nu\,\varepsilon_{30}=0\\ -\varepsilon_{29}\varepsilon_{28}...\varepsilon_{23}, & \alpha\nu\,\varepsilon_{30}=1\end{cases}$$

$$M=\begin{cases}0.\psi_{22}\psi_{21}...\psi_{0}, & \alpha\nu\,s_{31}=0\\ -0.\psi_{22}\psi_{21}...\psi_{0}, & \alpha\nu\,s_{31}=1\end{cases}$$

Ο εκθέτης μπορεί να παίρνει (ακέραιες) τιμές, από –127 μέχρι +127. Η ελάχιστη μη μηδενική τιμή της μαντίσας μπορεί να είναι $0.1_2 = 0.5_{10}$. Η μέγιστη σχηματίζεται όταν όλα τα ψηφία (22 - 0) είναι 1:

$2^{-1} + 2^{-2} + ... + 2^{-23} = 1 - 2^{-23} \approx 0.99999988 \approx 1$

Η ελάχιστη θετική τιμή που μπορεί να παρασταθεί είναι η:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Δηλαδή, ελάχιστη μη-μηδενική μαντίσα ($0.1_2 = 0.5_{10}$) και ελάχιστος εκθέτης (-127). Προφανώς πρόκειται για το $2^{-128} \approx 0.29387e-38$.

Η μέγιστη τιμή είναι η:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

που σημαίνει: μέγιστη μαντίσα (≈ 1) και μέγιστος εκθέτης (+127). Η τιμή είναι: $2^{127} \approx 0.17014e+39$.

Το 1 παριστάνεται ως:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

δηλαδή: $0.1_2 \times 2^1$.

Ο πλησιέστερος μεγαλύτερος αριθμός είναι αυτός που προκύπτει αν αλλάξουμε σε 1 το ψηφίο της θέσης 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Η διαφορά του από το 1 είναι $2^{-23} \times 2^1 \approx 0.23842e-06$. Αυτό είναι το ε για την παράσταση αυτή.

17.11.1 Πόλωση

Η μορφή πρόσημο - απόλυτη τιμή δεν είναι η καλύτερη δυνατή για τον εκθέτη:

- οι πράξεις είναι πολύπλοκες και
- περιορίζεται λιγάκι η περιοχή τιμών (έχουμε δυο παραστάσεις για το 0: +0, –0).

Ένα απλό τέχνασμα μας απαλλάσσει από τα παραπάνω: όταν αποθηκεύουμε μια τιμή, ο εκθέτης δεν αποθηκεύεται όπως είναι, αλλά αφού πρώτα του προσθέσουμε 127. Μέ τον τρόπο αυτόν ο εκθέτης είναι πάντοτε μη αρνητικός. Λέμε ότι ο εκθέτης είναι **πολωμένος** (biased) κατά 127.

Πρόσεξε ότι όταν κάνουμε πράξεις μεταξύ τιμών τύπου `float` δεν χρειάζεται να κάνουμε κάτι ιδιαίτερο λόγω της πόλωσης του εκθέτη. Τώρα οι πράξεις είναι πιο απλές (δεν υπάρχει πρόσημο) και οι τιμές του εκθέτη μπορούν να είναι από 0 μέχρι 255 ή πραγματικές τιμές –127 μέχρι 128. Έχουμε δηλαδή 256 διαφορετικές τιμές αντί για 255 που είχαμε χωρίς πόλωση.

Όταν όμως θέλουμε να τυπώσουμε τον αριθμό, θα πρέπει να αφαιρούμε από τον εκθέτη το 127. Στην παράσταση:

$$M \times 2^e$$

το νόημα του e αλλάζει:

$e = \varepsilon_{30}\varepsilon_{29}...\varepsilon_{23}$ – 1111111 $(1111111_2 = 127_{10})$.

Όταν έχουμε πόλωση,

- Η ελάχιστη θετική τιμή που μπορεί να παρασταθεί έχει τη μορφή:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  ελάχιστος εκθέτης: 0 –στην πραγματικότητα 0 – 127 = –127– και φυσικά ελάχιστη μη-μηδενική μαντίσα $(0.1_2 = 0.5_{10})$. Προφανώς πρόκειται για την τιμή που υπολογίσαμε και πριν: $2^{-128} \approx 0.29387e{-38}$.

- Η μέγιστη τιμή είναι η:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

  μέγιστος εκθέτης: 255 –με πραγματική τιμή 255 – 127 = 128– και μέγιστη μαντίσα (≈ 1). Η τιμή είναι: $2^{128} \approx 0.34028e{+39}$, δηλαδή διπλή από αυτήν που είχαμε χωρίς πόλωση.

- Το 1 παριστάνεται ως:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  δηλαδή: $0.1_2 \times 2^{128-127}$.

  Είναι φανερό ότι το ε δεν αλλάζει.

17.11.2 Άλλες Περιπλοκές – Πρότυπο IEEE

Μια άλλη παρατήρηση που μπορούμε να κάνουμε είναι η εξής: Αφού η τιμή του πρώτου ψηφίου της μαντίσας είναι πάντοτε 1, δεν υπάρχει λόγος να αποθηκεύεται· αυτό και γίνεται σε ορισμένες παραστάσεις. Βεβαίως, όπως είπαμε, είναι πάντοτε 1, εκτός από μια περίπτωση: όταν αποθηκεύουμε το 0. Θα πρέπει λοιπόν να κάνουμε ορισμένες συμβάσεις. Θα δούμε μερικές τέτοιες συμβάσεις που γίνονται στο πρότυπο της IEEE για αριθμητική κινητής υποδιαστολής, μια και το πρότυπο αυτό το συναντούμε σε πολλούς σύγχρονους υπολογιστές.

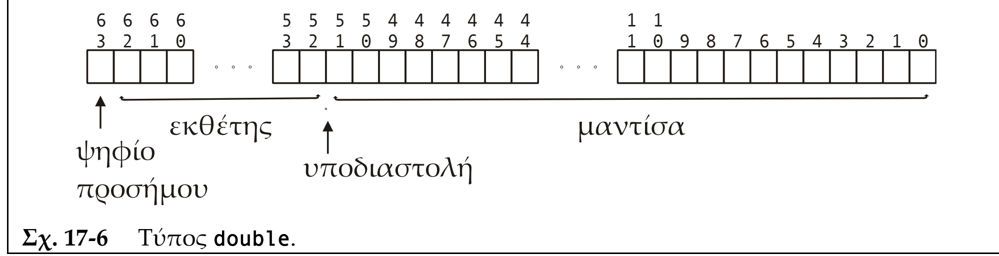
Χρησιμοποιούμε και πάλι 32 δυαδικά ψηφία (31 - 0), στο ψηφίο 31 έχουμε το πρόσημο και στα 30 - 23 έχουμε τον εκθέτη, πολωμένο κατά 127.

Για τη μαντίσα όμως τα πράγματα αλλάζουν: ακριβώς πριν από το ψηφίο 22 νοείται ότι υπάρχει το "`.1`". Έτσι, το νόημα της μαντίσας είναι:

$$M = \begin{cases} 0.1\psi_{22}\psi_{21}...\psi_0, & \alpha\nu\, s_{31}=0 \\ -0.1\psi_{22}\psi_{21}...\psi_0, & \alpha\nu\, s_{31}=1 \end{cases}$$

$e = \varepsilon_{30}\varepsilon_{29}...\varepsilon_{23}$ – 1111111 $(1111111_2 = 127_{10})$.

Το μηδέν παριστάνεται με 32 δυαδικά ψηφία ίσα με 0. Αυτή είναι μια *σύμβαση*, αφού σύμφωνα με όσα είπαμε ο αριθμός αυτός θα έπρεπε να είναι ίσος με: $1 \times 2^{-127} = 5.877 \times 10^{-39}$. Γενικώς, υπάρχει μια σύμβαση (IEEE) για τις παραστάσεις που έχουν όλα τα ψηφία του εκθέτη 0. Πρόκειται για **μη-κανονικοποιημένες** (denormalized ή subnormal) παραστάσεις (πολύ μικρών) αριθμών.



Σχ. 17-6 Τύπος `double`.

- Ο *μικρότερος (θετικός) κανονικοποιημένος αριθμός*, σύμφωνα με το πρότυπο, είναι αυτός που έχει 1 το λιγότερο σημαντικό ψηφίο (23) του εκθέτη και όλα τα άλλα 0:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0|0|0|0|0|0|0|0|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
```

  δηλαδή: $1 \times 2^{1-127} = 2^{-126} \approx 0.11755e$-37.

- Η παράσταση του μέγιστου θετικού είναι αυτή που έχει όλα τα ψηφία του εκθέτη, εκτός από το τελευταίο (23), ίσα με 1 και όλα τα ψηφία της μαντίσας 1:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0|1|1|1|1|1|1|1|0|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1
```

  Η τιμή αυτή είναι περίπου: $2 \times 2^{254-127} = 2^{128} \approx 0.34028e$+39.

- Το 1 παριστάνεται ως:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0|0|1|1|1|1|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
```

  ενώ ο πλησιέστερος μεγαλύτερος αριθμός είναι ο:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0|0|1|1|1|1|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1
```

  Το ε είναι $2^{-23} \approx 1.1921e$-07. Μπορούμε δηλαδή να παραστήσουμε 24 σημαντικά δυαδικά ψηφία, που ισοδυναμούν με 7 δεκαδικά ψηφία περίπου.

- Το **άπειρο** (`INF`) παριστάνεται ως εξής: όλα τα ψηφία του εκθέτη 1 και όλα τα ψηφία της μαντίσας 0.
- Αν όλα τα ψηφία του εκθέτη είναι 1 και ένα τουλάχιστον ψηφίο της μαντίσας δεν είναι 0, *η παράσταση θεωρείται ότι δεν παριστάνει αριθμό* (**N**ot **a** **N**umber, `NaN`).

17.11.3 Οι Τύποι "`double`" και "`long double`"

Ο τύπος `double` της C++ είναι «διπλός `float`» (§17.10). Θα δώσουμε τώρα μια παράσταση σύμφωνη με το πρότυπο της IEEE.

Όπως προαναφέραμε, ο χώρος που καταλαμβάνεται από μια θέση τύπου `double` καταλαμβάνει στη μνήμη διπλό χώρο απ' ότι μια θέση τύπου `float`. Αν λοιπόν αποθηκεύουμε μια τιμή τύπου `float` σε 32 δυαδικά ψηφία, μια τιμή τύπου `double` θα αποθηκεύεται σε 64 δυαδικά ψηφία, που τα αριθμούμε από 0 μέχρι 63 (Σχ. 17-6).

Ο εκθέτης αποθηκεύεται σε 11 δυαδικά ψηφία (62 - 52) πολωμένος κατά 1023, ενώ η μαντίσα αποθηκεύεται σε 52 ψηφία (51 - 0).

Ο *μικρότερος (θετικός) κανονικοποιημένος αριθμός*, σύμφωνα με το πρότυπο, είναι αυτός που έχει το λιγότερο σημαντικό ψηφίο (52) του εκθέτη ίσο με 1 και όλα τα άλλα 0: δηλαδή: $1 \times 2^{1-1023} = 2^{-1022} \approx 0.22251e$-307. Με μη κανονικοποιημένες μορφές μπορεί να παρασταθούν και μικρότεροι αριθμοί, μέχρι 5.0e-324.

Η παράσταση του μέγιστου ακέραιου είναι αυτή που έχει όλα τα ψηφία του εκθέτη, εκτός από το τελευταίο (52), ίσα με 1 και όλα τα ψηφία της μαντίσας 1: Η τιμή αυτή είναι περίπου: $2 \times 2^{2046-1023} = 2^{1024} \approx 0.17977e$+309

Το ε είναι $2^{-52} \approx 2.220446049250e$-16. Μπορούμε δηλαδή να παραστήσουμε 53 σημαντικά ψηφία στο δυαδικό σύστημα ή περίπου 16 σημαντικά ψηφία στο δεκαδικό.

Η C++ δίνει και τον τύπο `long double`, οι τιμές του οποίου αποθηκεύονται σε 10 ψηφιολέξεις, δηλ. 80 δυαδικά ψηφία, με 15ψήφιο εκθέτη και 64ψήφια μαντίσα. Δίνει ακρίβεια 20 (δεκαδικών) ψηφίων με τιμές από 1.9*e*-4951 μέχρι 1.1*e*4932.

Οι παραπάνω τύποι, `float`, `double`, `long double` δεν είναι αποκλειστικότητα της C++· θα τους βρεις και σε άλλες γλώσσες προγραμματισμού.

17.11.4 Μερικά Χρήσιμα Εργαλεία από τη C

Η C μας δίνει μερικές (μακρο)συναρτήσεις για να καταλαβαίνουμε τι συμβαίνει με παραστασεις τιμών κινητής υποδιαστολής. Οι ορισμοί τους υπάρχουν στο `cmath`.

`int fpclassify(` *τύπος κινητής υποδιαστολής* `x );`

όπου:

τύπος κινητής υποδιαστολής `= "float" | "double" | "long double" ;`

Η *fpclassify()* επιστρέφει ως τιμή μια από τις *αμοιβαίως αποκλειόμενες* τιμές `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO` που ορίζονται (με "`#define`") επίσης στο `cmath`.

Μας δίνει ακόμη τα εξής κατηγορήματα:

`int isinf(` *τύπος κινητής υποδιαστολής* `x );`
`int isnan(` *τύπος κινητής υποδιαστολής* `x );`
`int isnormal(` *τύπος κινητής υποδιαστολής* `x );`
`int isfinite(` *τύπος κινητής υποδιαστολής* `x );`

Σημείωση: ▶

Για να τις χρησιμοποιήσεις θα πρέπει να τις σκέφτεσαι ως

`bool isinf(` *τύπος κινητής υποδιαστολής* `x );`
`bool isnan(` *τύπος κινητής υποδιαστολής* `x );`
`bool isnormal(` *τύπος κινητής υποδιαστολής* `x );`
`bool isfinite(` *τύπος κινητής υποδιαστολής* `x );`

Έτσι, μπορείς να γράφεις "`if ( isnan(sqrt(x) ) …`"◀

Το *isnormal()* επιστρέφει "`1`" ("`true`") αν η *x* έχει κανονικοποιημένη παράσταση (ούτε "`0`", ούτε μη-κανονικοποιημένη, ούτε άπειρο, ούτε NaN).

Το *isfinite()* επιστρέφει "`1`" ("`true`") αν η *x* δεν είναι άπειρο ούτε NaN.

Για την (πεπερασμένη!) παράσταση του απείρου ορίζονται οι σταθερές `HUGE_VALF`, `HUGE_VAL`, `HUGE_VALL` για τους `float`, `double`, `long double` αντιστοίχως.[11]

Το παρακάτω προγραμματάκι χρησιμοποιεί τα παραπάνω εργαλεία:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   float x( 0 );
   double dn( 1e-100 ), dx( 1e100 );

   if ( fpclassify(x) == FP_ZERO )
      cout << "x is ZERO" << endl;
   cout << dn << "   " << isinf(dn) << "   " << isnan(dn)
        << "   " << isnormal(dn) << "   " << isfinite(dn) << endl;
   cout << dx << "   " << isinf(dx) << "   " << isnan(dx)
        << "   " << isnormal(dx) << "   " << isfinite(dx) << endl;
   x = dn;
   cout << x << "   " << isinf(x) << "   " << isnan(x)
        << "   " << isnormal(x) << "   " << isfinite(x) << endl;
```

[11] Στο πρότυπο C99 υπάρχει πρόβλεψη και για άλλο (πιο «πραγματικό») άπειρο, το `INFINITY`, για υλοποιήσεις που μπορούν να το υποστηρίξουν.

```
   x = dx;
   cout << x << "   " << isinf(x) << "   " << isnan(x)
        << "   " << isnormal(x) << "   " << isfinite(x) << endl;
   if ( x == HUGE_VALF )
      cout << "It IS HUGE!!!" << endl;
   cout << 1/INFINITY << endl;
}
```

και μας δίνει:

```
x is ZERO
1e-100  0  0  1  1
1e+100  0  0  1  1
0  0  0  0  1
1.#INF  1  0  0  0
It IS HUGE!!!
0
```

17.12 Σφάλμα από Μετατροπή Τύπου

Ξεκινάμε με ένα μικρό προγραμματάκι:

```
int main()
{
   long   i1, i2;
   float  r1, r2;

   i1 = 123456787;
   i2 = 123456786;
   r1 = i1;  r2 = i2;
   if ( r1 == r2)  cout << "Ε, ΛΟΙΠΟΝ ΕΙΝΑΙ ΙΣΑ!!!" << endl;
}
```

Μεταγλωττίζουμε το πρόγραμμα αυτό με τη Borland C++ όπου οι τιμές τύπων `long` και `float` παριστάνονται με 32 δυαδικά ψηφία, όπως είδαμε παραπάνω. Ακόμα, χειρίζεται την παράσταση κινητής υποδιαστολής σύμφωνα με το πρότυπο της IEEE. Το αποτέλεσμα; Νάτο:

```
Ε, ΛΟΙΠΟΝ ΕΙΝΑΙ ΙΣΑ!!!
```

Ας προσπαθήσουμε να βγάλουμε άκρη: Οι μεταβλητές *i*1 και *i*2 παίρνουν τις τιμές τους χωρίς κανένα πρόβλημα, μια και βρίσκονται μέσα στα όρια των τιμών που μπορούν να πάρουν (§4.3). Στην εντολή "`r1 = i1`" γίνονται τα εξής: πριν αποθηκευτεί η τιμή του *i*1 στην θέση της μνήμης *r*1 μετατρέπεται σε παράσταση κινητής υποδιαστολής. Αλλά, όπως είδαμε στην §4.3, μπορούμε να κρατήσουμε το πολύ 7 σημαντικά ψηφία. Έτσι, η «λεπτομέρεια» των δυο τελευταίων ψηφίων χάνεται. Το ίδιο συμβαίνει και με την εκτέλεση της "`r2 = i2`". Στην επόμενη εντολή –"`if (r1 == r2) cout <<...`"– όταν γίνεται η σύγκριση των *r*1 και *r*2, αυτές βρίσκονται ίσες αφού η σύγκριση γίνεται με τα πρώτα έξη ψηφία!

Φτάνουμε λοιπόν στο συμπέρασμα: *Κατά τη μετατροπή τύπου από ακέραιο τύπο σε τύπο κινητής υποδιαστολής μπορεί να έχουμε απώλεια σημαντικών ψηφίων.*

17.13 Τα Σφάλματα και πώς Μεταδίδονται

Αφού είδαμε τι συμβαίνει με την παράσταση στον τύπο `float` στο δεκαδικό σύστημα και στο δυαδικό, θα δούμε τώρα την παράσταση πιο γενικά· θα βγάλουμε ένα άνω φράγμα για το σχετικό σφάλμα από την παράσταση.

Στη συνέχεια θα δούμε πως μεταδίδονται τα σφάλματα από τους αριθμούς στις πράξεις μεταξύ τους.

17.13.1 Το Σφάλμα Παράστασης

Ας υποθέσουμε ότι έχουμε έναν υπολογιστή που δουλεύει σε σύστημα με βάση b, μπορεί να παραστήσει N ψηφία στο σύστημα αυτό. Το b είναι συνήθως 2, 8, 10, 16. Ας υποθέσουμε ότι η παράσταση ενός αριθμού x γίνεται στη μορφή:

$$x_f = M \times b^e$$

όπου:

$M = \pm(\psi_{-1}b^{-1} + \psi_{-2}b^{-2} \ldots \psi_{-N}b^{-N})$

$0 \le \psi_k \le b - 1, \quad k = 1 \,..\, N$

$E_E \le e \le E_M$

Όταν παριστάνουμε έναν αριθμό κρατάμε τα N πιο σημαντικά ψηφία του. Μπορούμε να πάρουμε μια τέτοια προσέγγιση είτε με **στρογγύλευση** (rounding) είτε με **αποκοπή** (truncation, choping). Να δούμε τι σφάλμα κάνουμε στις δυο περιπτώσεις.

Στρογγύλευση: Αν το ψηφίο (N-1) τάξης είναι $\ge b/2$ τότε το ψ_N αυξάνεται κατά μια μονάδα. Επομένως το (απόλυτο) σφάλμα που κάνουμε, $|x_f - x|$, είναι το πολύ:

$$|x_f - x| \le (b/2) \times b^{-(N+1)} \times b^e = 0.5 \times b^{e-N}$$

Από τη σχέση αυτή μπορούμε να υπολογίσουμε το μέγιστο **σχετικό σφάλμα** (relative error) λόγω της στρογγύλευσης:

$$\frac{|x_f - x|}{|x|} \le 0.5\frac{b^{e-N}}{|x|} = 0.5\frac{b^{e-N}}{(0.\psi_1\psi_2\ldots)b^e} = 0.5\frac{b^{-N}}{0.\psi_1\psi_2\ldots}$$

Αλλά, $0.\psi_1\psi_2\ldots_b \ge 0.100\ldots_b$ ($= b^{-1}$) και έτσι:

$$\text{Σχετ}(x_f) = \frac{|x_f - x|}{|x|} \le 0.5b^{-N+1}$$

Ας εφαρμόσουμε αυτή τη σχέση στις παραστάσεις που είδαμε στις προηγούμενες παραγράφους. Πρώτα στον δεκαδικό υπολογιστή: εδώ έχουμε $b = 10$, $N = 5$, άρα, για κάθε x:

$$\text{Σχετ}(x_f) \le 0.5\times10^{-4} \quad \text{ή Σχετ}(x_f) \le 0.00005$$

Στη δυαδική παράσταση της §14.7: $b = 2$, $N = 23$, άρα:

$$\text{Σχετ}(x_f) \le 0.5\times2^{-23}$$

Αλλά, $0.5\times2^{-23} = 2^{-24} \approx 10^{-7.22} \le 0.6\times10^{-7}$ και μπορούμε να γράψουμε:

$$\text{Σχετ}(x_f) \le 0.00000006$$

Αποκοπή: Στην περίπτωση αυτή αποκόπτονται όλα τα ψηφία μετά το N-οστό. Όλο το τμήμα που αποκόπτεται είναι μικρότερο από μια μονάδα της τάξης του N-οστού ψηφίου. Δηλαδή:

$$|x_f - x| < 1\times b^{e-N}$$

Έτσι, το φράγμα του σχετικού σφάλματος είναι διπλάσιο αυτού που είχαμε στην περίπτωση της στρογγύλευσης:

$$\text{Σχετ}(x_f) = \frac{|x_f - x|}{|x|} < b^{-N+1}$$

17.13.2 Μετάδοση Σφαλμάτων

Έχουμε δυο τιμές x, y και τις παριστάνουμε στον υπολογιστή μας ως x_f, y_f σε παράσταση κινητής υποδιαστολής. Θα έχουμε:

$$x = x_f + \delta x \qquad y = y_f + \delta y$$

Ας συμβολίσουμε με π μια από τις πράξεις +, –, ×, / και με π_f την αντίστοιχη πράξη του υπολογιστή μας, μια από τις: $+_f$, $-_f$, $*_f$, $/_f$. Για το σφάλμα του αποτελέσματος θα έχουμε:

$(x \,\pi\, y) - (x_f \,\pi_f\, y_f) = (x \,\pi\, y) - (x_f \,\pi_f\, y_f) + (x_f \,\pi\, y_f) - (x_f \,\pi\, y_f)$
$= (x_f \,\pi\, y_f - x_f \,\pi_f\, y_f) + (x \,\pi\, y - x_f \,\pi\, y_f)$

Ο πρώτος όρος αναφέρεται μόνο σε τιμές που παριστάνονται ήδη στον υπολογιστή. Μας δίνει το σφάλμα που γίνεται από την *πράξη* π_f. Αν γυρίσεις στην §17.8 μπορείς να δεις πώς κάναμε στον δεκαδικό υπολογιστή την πρόσθεση των 14.563 (= x_f) και 0.16773 (= y_f). Ο πρώτος όρος για την περίπτωση αυτή είναι:

$(x_f + y_f) - (x_f +_f y_f) = (14.563 + 0.16773) - (14.563 +_f 0.16773)$
$= 14.73073 - 14.731$
$= -0.0008$

Αν υποθέσουμε ότι, όπως λέγαμε στην §14.8, η Αριθμητική Μονάδα κάνει τις πράξεις με μεγαλύτερη ακρίβεια και το αποτέλεσμα στρογγυλεύεται (ή αποκόπτεται) στη συνέχεια, μπορούμε να πούμε ότι:

$$(x_f \,\pi_f\, y_f) = (x_f \,\pi\, y_f)_f$$

Οπότε, σύμφωνα με αυτά που είπαμε για το σφάλμα στρογγύλευσης, στην προηγούμενη παράγραφο, έχουμε:

$$|x_f \,\pi\, y_f - x_f \,\pi_f\, y_f| \le 0.5\, |x_f \,\pi\, y_f|\, b^{1-N}$$

Ας έρθουμε τώρα στο δεύτερο όρο:

$$(x \,\pi\, y - x_f \,\pi\, y_f)$$

Εδώ *οι πράξεις είναι ακριβείς*, αλλά συγκρίνουμε το αποτέλεσμα που παίρνουμε από τις αληθείς τιμές με αυτό που παίρνουμε από τις (προσεγγιστικές) τιμές σε μορφή κινητής υποδιαστολής. Αυτό λέγεται **μεταδιδόμενο σφάλμα** (propagated error). Θα δούμε στη συνέχεια το μεταδιδόμενο σφάλμα για τις τέσσερις πράξεις.

Πολλαπλασιασμός:

$xy - x_f y_f = xy - (x - \delta x)(y - \delta y)$
$= x\delta y + y\delta x - \delta x \delta y$

και

$$\Sigma\chi\epsilon\tau(x_f y_f) \le \Sigma\chi\epsilon\tau(x_f) + \Sigma\chi\epsilon\tau(y_f) + \Sigma\chi\epsilon\tau(x_f)\Sigma\chi\epsilon\tau(y_f)$$

Αν $\Sigma\chi\epsilon\tau(x_f)\Sigma\chi\epsilon\tau(y_f) \ll 1$ τότε:

$$\Sigma\chi\epsilon\tau(x_f y_f) \le \Sigma\chi\epsilon\tau(x_f) + \Sigma\chi\epsilon\tau(y_f)$$

Διαίρεση:

$$\Sigma\chi\epsilon\tau\left(\frac{x_f}{y_f}\right) = \frac{|x/y - x_R/y_R|}{|x/y|} = \frac{|y_R/y - x_R/x|}{|y_R/y|}$$

$$= \frac{|\delta x/x - \delta y/y|}{|1 - \delta y/y|} \le \frac{\Sigma\chi\epsilon\tau(x_f) + \Sigma\chi\epsilon\tau(y_f)}{|1 - \Sigma\chi\epsilon\tau(y_f)|}$$

Και εδώ, αν $\Sigma\chi\epsilon\tau(x_f)\Sigma\chi\epsilon\tau(y_f) \ll 1$ τότε:

$$\Sigma\chi\epsilon\tau(x_f y_f) \le \Sigma\chi\epsilon\tau(x_f) + \Sigma\chi\epsilon\tau(y_f)$$

Όπως βλέπεις, στον πολλαπλασιασμό και τη διαίρεση τα σφάλματα μεταδίδονται αργά. Είναι, θα μπορούσαμε να πούμε, αρκετά «σίγουρες» πράξεις. Δεν συμβαίνει όμως το ίδιο με τις άλλες δυο πράξεις που έχουμε αφήσει τελευταίες.

Πρόσθεση - Αφαίρεση: Αν χρησιμοποιήσουμε το π για το + ή το -, το απόλυτο σφάλμα (που αντιστοιχεί στο δεύτερο όρο) είναι:

$x \,\pi\, y - x_f \,\pi\, y_f = x \,\pi\, y - (x - \delta x) \,\pi\, (y - \delta y)$
$= \delta x \,\pi\, \delta y$
$\le |\delta x| + |\delta y|$

Ας δούμε τι σημαίνει αυτό, με ένα παράδειγμα: έχουμε τους αριθμούς 0.123445 και 0.123454 και χρησιμοποιούμε το δεκαδικό μας υπολογιστή για να υπολογίσουμε τη διαφορά

τους. Ο πρώτος παριστάνεται ως: 0.12345 (σφάλμα -0.000005) και ο δεύτερος: 0.12345 (σφάλμα 0.000004) και η διαφορά τους υπολογίζεται: 0. Το ακριβές αποτέλεσμα είναι 0.000009 και το σχετικό σφάλμα είναι 100%! Για να δούμε τι έγινε εδώ: Έχουμε δυο αριθμούς με έξη ψηφία και παραπλήσιες απόλυτες τιμές –η διαφορά τους βρίσκεται στα δυο τελευταία ψηφία. Για να παρασταθούν οι αριθμοί στον υπολογιστή, που δέχεται μόνο 5 ψηφία, γίνεται το καλύτερο δυνατό: στρογγύλευση σε 5 ψηφία. Τώρα, τα τέσσερα πρώτα –και σωστά– ψηφία είναι ίδια ενώ τα 5α ψηφία είναι λάθος, λόγω της στρογγύλευσης. Το αποτέλεσμα υπολογίστηκε ακριβώς με βάση τα 5α ψηφία.

Μήπως αυτό ήταν ένα «παρατραβηγμένο» παράδειγμα; Πόσο συχνά συμβαίνει κάτι τέτοιο; Σχεδόν κάθε φορά που έχουμε να αφαιρέσουμε δυο αριθμούς με παραπλήσιες απόλυτες τιμές. Οι αριθμοί αυτοί συνήθως προκύπτουν από (μη ακριβείς) πράξεις και έχουν κάποιο σφάλμα. Το σφάλμα αυτό βρίσκεται στα τελευταία (λιγότερο σημαντικά) ψηφία. Έτσι, με μια αφαίρεση, παίρνουμε αποτέλεσμα που μπορεί να είναι τελείως λάθος, όπως στο παράδειγμά μας.

17.14 Ισότητα στους Τύπους Κινητής Υποδιαστολής

Τα παραπάνω γεννούν και ένα άλλο ερώτημα: αν έχουμε δηλώσει:

```
float A, B;
```

πόσο νόημα έχει η σύγκρισή τους για ισότητα:

```
if ( A == B ) ...
```

Ας πούμε α, β τις ακριβείς τιμές των ποσοτήτων που φιλοδοξούμε να υπολογίσουμε στο πρόγραμμά μας με τα A, B αντίστοιχα. Οι πράξεις που θα κάνουμε στον υπολογιστή μας θα εισαγάγουν σφάλματα και τελικώς:

$$|A - \alpha| \leq \delta A \quad \text{και} \quad |B - \beta| \leq \delta B$$

ή

$$\alpha - \delta A \leq A \leq \alpha + \delta A \quad \text{και} \quad \beta - \delta B \leq B \leq \beta + \delta B$$

Από αυτές παίρνουμε:

$$(\alpha - \beta) - (\delta A + \delta B) \leq A - B \leq (\alpha - \beta) + (\delta A + \delta B)$$

Αυτό που μας ενδιαφέρει βεβαίως, είναι να συγκρίνουμε για ισότητα τα α και β. Αν $\alpha = \beta$ τότε για τα A και B έχουμε:

$$-(\delta A + \delta B) \leq A - B \leq (\delta A + \delta B)$$

Όπως φαίνεται λοιπόν, το σωστό είναι να ρωτήσουμε:

```
if ( fabs(A - B) <= eps ) ...
```

Το *eps*, κατ' αρχήν, εξαρτάται από

- το συγκεκριμένο πρόβλημα, αφού εξαρτάται από τα σφάλματα παράστασης και πράξεων που συσσωρεύτηκαν κατά τους υπολογισμούς των A και B.
- την ακρίβεια που απαιτούμε στον υπολογισμό του A ή του B.

Αλλά, το *eps* έχει περιορισμούς από τον τύπο `float` που δουλεύουμε: *δεν μπορεί να είναι μικρότερο* από το

$$\delta A = min\{\ u\colon \mathbb{R} \mid u > 0 \wedge (|A|+u)_R \neq |A|_R \bullet u\ \}$$

Αποδεικνύεται ότι για το δA έχουμε:

$$|A|\varepsilon/b < \delta A \leq |A|\varepsilon$$

όπου b η βάση του αριθμητικού συστήματος του υπολογιστή σου και ε το *έψιλον* του τύπου `float`. Αν έχεις $b = 2$, τότε $|A|\varepsilon/2 < \delta A \leq |A|\varepsilon$.

Από τον ορισμό έχεις ότι: αν $0 \leq x < \delta A$ τότε αν $B = A + x$ θα πρέπει να δεχτείς ότι A `==` B αφού ο υπολογιστής σου δεν μπορεί να σου δώσει κάτι ακριβέστερο· όλοι οι B, με τα παραπάνω χαρακτηριστικά, παριστάνονται με τον ίδιο τρόπο με τον A. Δηλαδή, αν ήδη δεν μπορείς να ξεχωρίσεις δυο αριθμούς που διαφέρουν λιγότερο από δA, λόγω του τρόπου

που παριστάνονται στον τύπο **float**, δεν έχει νόημα να προσπαθείς να ανιχνεύσεις διαφορά παράστασης μικρότερη από δ*A*. Το *eps* λοιπόν δεν έχει νόημα αν είναι μικρότερο από δ*A* και, επειδή αυτό που έχουμε εύκολα είναι το πάνω φράγμα του, θα πρέπει $eps \geq |A|\varepsilon$.

«Δεν έχει νόημα» ή θα έχουμε πρόβλημα; Μπορεί να έχεις και πρόβλημα! Για παράδειγμα, στις επαναπροσεγγιστικές (iterative) μεθόδους συχνά δουλεύουμε ως εξής: βρίσκουμε ένα διάστημα [*a*, *b*] μέσα στο οποίο βρίσκεται η τιμή ξ που θέλουμε να προσεγγίσουμε και μικραίνουμε το διάστημα έτσι ώστε να γίνει τελικά μηδενικού μήκους, οπότε θα έχουμε $a=\xi=b$. Σύμφωνα μεαυτά που είπαμε, θα κάνουμε έλεγχο ως εξής:

```
while ( fabs(b-a) > eps )...
```

αλλά, θα έλεγε κανείς, εδώ πέρα το *eps* ας είναι ό,τι θέλει, ακόμη και μηδέν. Το πολύ-πολύ να γίνουν τα *a* και *b* ίσα, πράγμα όχι άσχημο. Λοιπόν: *δεν είναι έτσι*. Μπορεί (αυτό εξαρτάται από τη μέθοδο) το *b* να γίνει $a+\delta A$. Από κει και πέρα οι τιμές των *a* και *b* (επιλεγόμενες από το [*a*,*b*]) μπορεί να μην αλλάζουν, ενώ $fabs(b-a) = \delta A > eps$. Μια τέτοια μέθοδος είναι αυτή της διχοτόμησης και το πρόβλημα φαίνεται στο παρακάτω

Παράδειγμα

Από την §14.3 αντιγράφουμε, με μικρές αλλαγές, το παρακάτω πρόγραμμα, που δοκιμάζει μια συνάρτηση για επίλυση αλγεβρικών εξισώσεων με τη μέθοδο της διχοτόμησης.

```
#include <iostream>
#include <cmath>

using namespace std;

double q( double x );
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                double& root, int& errCode );

int main()
{
   double riza;
   int    errCode;

   bisection( q, 0.1, 1.0, 1e-20, riza, errCode );
   if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                  else  cout << " Ρίζα = " << riza << endl;
} // main
```

Δεν έχουμε αλλάξει τίποτε στη *bisection*· αλλά, ενώ στήν §14.3 την είχαμε καλέσει με *epsilon* = 1*e*-5 –και μας έδωσε προσέγγιση της ρίζας 0.1585983– τώρα την καλούμε με *epsilon* = 1e-20. Ο τύπος **double**, στην C++ που δουλεύουμε, είναι αυτός που είδαμε στην §14.7.3, με $\varepsilon = 2^{-23} \approx 2.220446049250e\text{-}16$.

Το πρόγραμμα, κατά την εκτέλεση, πέφτει σε αέναη ανακύκλωση. Ζητώντας εκτύπωση ενδιαμέσων στοιχείων βλέπουμε τα εξής:

#Επαν	a	b	|b-a|/2	m
:	:	:	:	:
54	0.15859434	0.15859434	5.55111512312578270e-17	0.15859434
55	0.15859434	0.15859434	2.77555756156289135e-17	0.15859434
56	0.15859434	0.15859434	1.38777878078144568e-17	0.15859434
57	0.15859434	0.15859434	1.38777878078144568e-17	0.15859434
58	0.15859434	0.15859434	1.38777878078144568e-17	0.15859434
59	0.15859434	0.15859434	1.38777878078144568e-17	0.15859434
:	:	:	:	:

Δηλαδή, το $m = (a+b)/2$ δεν μπορεί να γίνει καλύτερο και το $fabs(b-a)/2$ παραμένει μεγαλύτερο από το *epsilon*. Το πρόβλημα ξεκινάει από την τιμή του *epsilon*, που είναι πολύ μικρή για τον τύπο κινητής υποδιαστολής (**double**) που δουλεύουμε. Πράγματι,

$$|a|\varepsilon = 0.15859434 \times 2.220446049250e\text{-}16 \approx 3.521501756864e\text{-}17$$

και $\delta A \in [1.76e\text{-}17, 3.52e\text{-}17]$. Το *epsilon* δεν θα έπρεπε να είναι μικρότερο από δA.[12]

Και τί κάνουμε τώρα; Πώς διορθώνουμε το πρόγραμμα; Μια διόρθωση θα ήταν η εξής: Να βάλουμε ένα

$$trEps = max(|m|\varepsilon/b, epsilon)$$

και να αλλάξουμε τη συνθήκη της while σε `fabs(b-a)/2 >= trEps`. Στην περίπτωση αυτήν όμως η διαδικασία μας εξαρτάται από χαρακτηριστικά του τύπου `float` και χάνει τη δυνατότητα μεταφοράς. Καταφεύγοντας στην καλύτερη μελέτη της μεθόδου, βλέπουμε ότι $m_{νέο} = m_{παλιό} \pm |b - a|/2$. Αν λοιπόν το $|b - a|/2$ γίνει πολύ μικρό –*πρακτικώς μηδέν*– ως προς το *m*, δεν έχει νόημα να συνεχίζουμε. Πώς ελέγχεται αυτό σε C++; Με τον εξής «περίεργο» τρόπο:

```
m == m + fabs(b-a)/2
```

Εκτός από αυτό, η διαδικασία χρειάζεται και μια άλλη βελτίωση: το πρόγραμμα που την καλεί θα πρέπει να μπορεί να καθορίσει μέγιστο πλήθος επαναλήψεων που θα εκτελεσθούν, άσχετα με το αν πετύχαμε την ακρίβεια που θέλουμε:

```
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode )
{
   double m, d;

   if ( f(a)*f(b) > 0 )
      errCode = 3;
   else
   {
      int n( 0 );
      do {
         ++n;
         m = (a + b) / 2;
         if ( f(a)*f(m) <= 0.0 )  b = m;
                            else  a = m;
         d = fabs(b - a)/2;
      } while ( (d >= epsilon) && (m != m + d) && (n != nMax) );
      root = m;
      if (d <= epsilon)  errCode = 0;
      else if (m == (m + d))  errCode = 1;
      else if (n == nMax)  errCode = 2;
   } // if
} // bisection
```

Τώρα, που η επαναλήψεις σταματούν για τρεις διαφορετικούς λόγους, θα πρέπει να ελέγξουμε το λόγο τερματισμού και να τον γνωστοποιήσουμε στο πρόγραμμα που κάλεσε τη διαδικασία. Αυτό γίνεται με τη *errCode*:

errCode == 1 σημαίνει: Δεν μπορεί να γίνει άλλη βελτίωση.

errCode == 2 σημαίνει: Ξεπεράσαμε τις *nMax* επαναλήψεις χωρίς να επιτευχθεί η ακρίβεια που ζητήθηκε.

errCode == 3 σημαίνει: Λάθος αρχικό διάστημα.

Και να μια δοκιμή:

```
   bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
```

αποτέλεσμα:

[12] Όπως καταλαβαίνεις, τα τελευταία ψηφία των *a*, *b* δεν έχουν νόημα.

```
Λάθος 1
Ρίζα =  0.158594
Λάθος 2
Ρίζα =  0.184375
Λάθος 0
Ρίζα =  0.158594
```

Με την ευκαιρία, πρόσεξε και κάτι άλλο: στον έλεγχο της *n* δεν βάλαμε *n* `<` *nMax* αλλά *n* `!=` *nMax*· το αποτέλεσμα; Αν η συνάρτηση κληθεί με *nMax* ≤ 0 ο αριθμός των επαναλήψεων δεν ελέγχεται.

Με τον ίδιο τρόπο που γράψαμε το «αν η βελτίωση είναι 0 ως προς το *m*» θα πρέπει να χειρίζεσαι πολλές φορές την περίπτωση: «αν το *A* είναι 0 τότε...». Αντί να γράφεις:

```
if ( fabs(A) <= eps ) ...
```

συχνά είναι προτιμότερο να γράφεις

```
if ( x + A = x ) ...
```

δηλαδή, αν το *A* είναι 0 ως προς το *x*. Φυσικά, το *x* εξαρτάται από το πρόβλημα που έχεις να λύσεις και πρέπει να επιλεγεί προσεκτικά.

17.15 Πρακτικές Συμβουλές

Οι πρώτες δυο συμβουλές για κάποιον που γράφει προγράμματα για επεξεργασία αριθμητικών στοιχείων είναι μάλλον τετριμμένες, αλλά πρέπει να τις πούμε:

- ***Μη χρησιμοποιείς χωρίς λόγο τους τύπους κινητής υποδιαστολής. Προτίμησε ακέραιους τύπους όπου μπορείς. Πρόσεχε, όμως τους πολύ μεγάλους ακέραιους!***
- ***Βελτίωσε τον αλγόριθμό σου ώστε να ελαττώσεις τις πράξεις που κάνεις στο ελάχιστο δυνατό!***

Καλά, θα πεί ο πεπειραμένος προγραμματιστής, για νέο μας το λες; Αυτό ισχύει σε κάθε περίπτωση. Σωστά! Αλλά, αν είναι στόχος να ελαττώνουμε τον χρόνο επεξεργασίας στο ελάχιστο για κάθε πρόγραμμα, για τα αριθμητικά προγράμματα έχουμε ένα λόγο ακόμη: *λιγότερες πράξεις σημαίνει και πιο ακριβές αποτέλεσμα*.

- ***Πρόσεξε τις προσθέσεις και τις αφαιρέσεις! Αν νομίζεις ότι μπορεί να σου δημιουργήσουν προβλήματα προσπάθησε να τις αποφύγεις!***

Να τις αποφύγω; Αν πρέπει να τις κάνω, πως να τις αποφύγω; Δυο παραδείγματα σου δίνουν μια ιδέα. Και τα δυο προγράμματα εκτελέστηκαν σε τύπο `float` 32 ψηφίων.

Παράδειγμα 1

Θέλουμε να λύσουμε την εξίσωση: $ax^2 + bx + c = 0$ όπου $a = 1.0$, $b = 1000000000.0 = 10^9$, $c = 4.0$ και φυσικά χρησιμοποιούμε τους γνωστούς μας τύπους:

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Ο τύπος `float` δουλεύει όπως περιγράψαμε στην §14.6.2. 'Οταν υπολογίζουμε το υπόριζο $b^2 = 10^{18}$, ενώ $4ac = 16$. Η αφαίρεση στον υπολογιστή μας θα δώσει αποτέλεσμα 10^{18} και τελικά θα έχουμε $x_+ = -1000000000.0$ και $x_- = 0$. Η πρώτη ρίζα προσεγγίστηκε θαυμάσια. Στη δεύτερη, η προσέγγιση $\sqrt{b^2 - 4ac} \approx b$ και η πράξη $-b + b$ (υποχείλιση) που ακολούθησε, μας έδωσε ένα τελείως λάθεμένο αποτέλεσμα. Τον υπολογισμό του υπορίζου δεν μπορούμε να τον βελτιώσουμε· ας τον χρησιμοποιήσουμε λοιπόν μόνο για την πρώτη ρίζα και ας αποφύγουμε την αφαίρεση. Θα στηριχτούμε στην ιδιότητα: $x_+x_- = c/a$, από όπου παίρνουμε: $x_- = c/(ax_+)$.

Το παρακάτω πρόγραμμα υπολογίζει το x_+(x1) και στη συνέχεια το x_- (x2) με τον γνωστό τύπο. Στη συνέχεια υπολογίζει τη δεύτερη ρίζα ως x3 χωρίς να κάνει την αφαίρεση.

```
int main()
{
   float a, b, c, x1, x2, x3;

   a = 1.0f;  b = 1000000000.0f;  c = 4.0f;
   x1 = (-b - sqrt(b*b - 4*a*c))/(2*a);
   x2 = (-b + sqrt(b*b - 4*a*c))/(2*a);
   x3 = c/(a*x1);
   cout << x1 << "  " << x2 << "  " << x3 << endl;
}
```

Να το αποτέλεσμα:

```
-1e+009  0  -4e-009
```

Παρατηρήσεις ▶

Για πρώτη φορά αναφέραμε ότι «*η αφαίρεση είναι μια «κακή» πράξη για τους τύπους κινητής υποδιαστολής διότι μπορεί να προκαλέσει απώλεια σημαντικών ψηφίων*» στην παρατήρηση 3 του Παραδ. 2 της §5.5. Μπορείς να κάνεις αυτά που λέμε παραπάνω με τα προγράμματα που είδαμε εκεί αφού αλλάξεις τον `double` σε `float`.◀

❖❖❖

Παράδειγμα 2 ✍

Θέλουμε να υπολογίσουμε την τιμή της παράστασης 1 - *συνt* για πολύ μικρά *t*. Αν δεν προσέξουμε θα πάρουμε 0 (υποχείλιση). Ας αποφύγουμε την αφαίρεση:

$$1-\sigma\upsilon\nu t = (1-\sigma\upsilon\nu t)\frac{1+\sigma\upsilon\nu t}{1+\sigma\upsilon\nu t} = \frac{\eta\mu^2 t}{1+\sigma\upsilon\nu t}$$

Και δοκιμάζουμε τους δυο τρόπους:

```
   t = 0.0000000001;
   x1 = 1 - cos(t);
   cout << x1 << endl;
   x2 = sin(t)*sin(t)/(1+cos(t));
   cout << x2 << endl;
```

αποτέλεσμα:

```
0
5e-021
```

Δηλαδή, με την ασφαλέστερη "1+*συνt*" αποφύγαμε την «καταστροφική» "1-*συνt*".

❖❖❖

♦ ***Απόφυγε την υπερχείλιση χωρίς λόγο.***

Όπως είδαμε πιο πριν, για τους τύπους κινητής υποδιαστολής, δεν ισχύουν η προσεταιριστικότητα της πρόσθεσης και του πολλαπλασιασμού, όπως και η επιμεριστικότητα των δυο πράξεων. Είναι πολύ πιθανό, με μια αναδιάταξη πράξεων σε μια αριθμητική παράσταση, να αποφύγεις μια υπερχείλιση από ενδιάμεσες πράξεις.

Ας δούμε δυο παραδείγματα:

Παράδειγμα 3 ✍

Από τα πρώτα πράγματα που μαθαίνει κανείς στον προγραμματισμό είναι να μην διαιρεί δια 0. Έτσι, η εντολή:

```
z = x/y;
```

αντικαθίσταται πολλές φορές από κάτι σαν:

```
if ( y != 0 )  z = x/y;
          else  ...
```

Όπως είπαμε πιο πάνω, κάτι τέτοιο δεν έχει και πολύ νόημα. Πιο σωστό θα ήταν κάτι σαν:

```
if ( fabs(y) > eps ) ...
```

Πράγματι, αν η τιμή του *y* έχει προκύψει μετά από πολλές πράξεις στον τύπο `float`, είναι απίθανο να είναι 0 όταν την περιμένουμε. Στην περίπτωση αυτήν ένα «απρόσεκτο» πρόγραμμα μπορεί να υποστεί τις συνέπειες μιας «κρυφής υπερχείλισης». Δηλαδή, το *y* δεν είναι ακριβώς μηδέν, αλλά έχει μια πολύ μικρή τιμή χωρίς κανένα νόημα. Το αποτέλεσμα θα είναι, πολλές φορές, να μην έχουμε υπερχείλιση αλλά το *z* να πάρει μια απίθανη τιμή που αλλοιώνει όλους τους υπολογισμούς στη συνέχεια.

Παράδειγμα 4

Παρόμοιες προφυλάξεις μπορούμε να πάρουμε και στην πρόσθεση. Ας υποθέσουμε ότι οι *x* και *y* μπορούν να πάρουν μεγάλες θετικές τιμές και η εντολή:

```
z = x + y;
```

θα μπορούσε να οδηγήσει σε υπερχείλιση. Θα μπορούσαμε και εδώ να προλάβουμε μια τέτοια πρόσθεση. Όπως ξέρουμε η C++ μας δίνει για τον τύπο `float` τη σταθερά *FLT_MAX* που είναι η μέγιστη τιμή που μπορεί να παρασταθεί στον τύπο `float`[13]. Μπορούμε λοιπόν, όπου χρειάζεται, να αντικαταστήσουμε την εντολή εκχώρησης με κλήση κάποιας διαδικασίας *addFloat*, που γράφεται όπως η *addInt* που γράψαμε για τον τύπο `int`.

Φυσικά, παρόμοιες προφυλάξεις μπορούμε να πάρουμε για όλες τις πράξεις και να έχουμε το κεφάλι μας ήσυχο. Ε, όχι κι έτσι! Ένα τέτοιο πρόγραμμα θα ήταν *απαράδεκτο*. Οι περισσότεροι από τους ελέγχους, που είναι *χρονοβόροι*, θα ήταν άχρηστοι. Ο *αμυντικός προγραμματισμός* σε σχέση με τις αριθμητικές πράξεις μπορεί να χρησιμοποιείται, αλλά μόνον όπου χρειάζεται.

Η τελευταία μας συμβουλή δεν μπορεί να είναι άλλη:

- ***Διάβασε Αριθμητική Ανάλυση.***

Με όσα είπαμε στις προηγούμενες παραγράφους, δεν φιλοδοξούμε να καλύψουμε τα θέματα που παρουσιάσαμε, αλλά περισσότερο να σου δείξουμε τα προβλήματα: *Τα αριθμητικά προγράμματα είναι δύσκολα. Η σύνθεσή τους απαιτεί* ***γνώσεις*** *και πολλή προσοχή.* Τις περισσότερες από τις γνώσεις που θα σου χρειαστούν θα σου τις δώσει η Αριθμητική Ανάλυση.

Ασκήσεις

Α Ομάδα

17-1 (*γενίκευση της count*1) Γράψε περίγραμμα συνάρτησης *xCount*1 που θα τροφοδοτείται με τιμή *x*, ακέραιου τύπου και δύο φυσικούς $k_1 \le k_2$ και θα υπολογίζει και θα επιστρέφει το πλήθος των δυαδικών ψηφίων που έχουν τιμή 1 στις θέσεις από k_1 μέχρι k_2 της *x*.

17-2 (*κυκλική ολίσθηση*) Γράψε συνάρτηση, με το όνομα *rShiftRight*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με μια τιμή *x*, τύπου `unsigned char`, και μια μη αρνητική ακέραιη τιμή *p* και θα υπολογίζει και θα επιστρέφει την *κυκλική* ολίσθηση δεξιά της *x* κατά *p* θέσεις (τα *p* δυαδικά ψηφία που χάνονται προς τα δεξιά εμφανίζονται από τα αριστερά, αντί μηδενικών, με την ίδια σειρά), όπως φαίνεται στο παρακάτω παράδειγμα. Αν η συνάρτησή μας κληθεί με το *x* που φαίνεται και $p = 3$ παίρνουμε το αποτέλεσμα που βλέπεις:

---

[13] Παρομοίως υπάρχουν και οι *DBL_MAX*, *LDBL_MAX* για τους τύπους `double` και `long double` αντίστοιχως.

x *αποτέλεσμα*

Μετάτρεψε τη συνάρτηση σε περίγραμμα συνάρτησης που θα δουλεύει για οποιονδήποτε ακέραιο τύπο.

Γράψε περίγραμμα συνάρτησης *rShiftLeft*, για κυκλική ολίσθηση αριστερά.

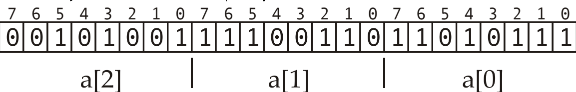
Β Ομάδα

17-3 (*ολίσθηση σε πίνακα*) Γράψε συνάρτηση, με το όνομα *shiftLeft*, που θα τροφοδοτείται, μέσω των παραμέτρων της, με έναν μονοδιάστατο πίνακα a, με στοιχεία τύπου `unsigned char`, το πλήθος των στοιχείων του *n* και μια μη αρνητική ακέραιη τιμή *p* και θα κάνει ολίσθηση αριστερά κατά *p* θέσεις των δυαδικών ψηφίων *ολόκληρου του πίνακα*, όπως φαίνεται στο παρακάτω παράδειγμα. Έχουμε αρχικώς ($n = 3$):

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1

a[2] | a[1] | a[0]

και ζητείται ολίσθηση κατά $p = 4$. Θα πάρουμε:



a[2] | a[1] | a[0]

Μετάτρεψε τη συνάρτηση σε περίγραμμα συνάρτησης που θα δουλεύει για οποιονδήποτε ακέραιο τύπο.

Γράψε περίγραμμα συνάρτησης *shiftRight*, για ολίσθηση πίνακα δεξιά.

17-4 Γενίκευσε τις *bitValue()*, *setBit()*, *clearBit()*, *count1()* και *part()* για πίνακα ακέραιου τύπου *T*. Αν κάθε τιμή τύπου *T* αποθηκεύεται σε s_T ψηφιολέξεις τότε η παράμετρος *p* των τριών πρώτων θα μπορεί να παίρνει τιμές από 0 μέχρι $n \times s_T - 1$, όπου *n* το πλήθος των στοιχείων του πίνακα.

17-5 Ακολουθώντας το παράδειγμα που δώσαμε στην §17.3 για την πρόσθεση, γράψε διαδικασίες για ασφαλείς πράξεις: αφαίρεση (*subtrInt*), πολλαπλασιασμό (*multInt*), διαίρεση (*divInt*) στον τύπο `int`.

17-6 Γράψε διαδικασίες για ασφαλείς πράξεις για τον τύπο `float`.

Γ Ομάδα

17-7 Αν *x* τιμή τύπου `float`, ορίζουμε:

$$\varepsilon_x = min\{ u{:}\ \mathbb{R} \mid u > 0 \wedge (|x|+u)_f \neq |x|_f \bullet u \}$$

Μπορείς να βρεις σχέση του ε_x με το ε; Πόσο σωστό είναι το $\delta x = x \cdot \varepsilon$;

[Απάντηση: $x \cdot \varepsilon / βάση < \varepsilon_x \leq x \cdot \varepsilon$]