κεφάλαιο

1

Υπολογισμοί με Σταθερές

Ο στόχος μας σε αυτό το κεφάλαιο:

Να γράψουμε πρόγραμμα που θα δουλεύει! Μόνο με σταθερές; Ναι! Μόνο με σταθερές, αλλά θα έχει μερικά βασικά χαρακτηριστικά που έχουν όλα τα προγράμματα.

Προσδοκώμενα αποτελέσματα:

Τελικώς θα μπορείς να γράψεις προγραμματάκια που θα κάνουν υπολογισμούς και θα βγάζουν τα αποτελέσματα μαζί με περιγραφικά κείμενα. Θα μάθεις όμως ότι για να γράψεις ένα πρόγραμμα C++ θα πρέπει να «στήσεις ένα περιβάλλον» μέσα στο οποίο θα βάλεις τις εντολές για τους υπολογισμούς.

Έννοιες κλειδιά:

- *εντολή εξόδου*
- *cout*
- *endl*
- *ορμαθοί χαρακτήρων*
- *αριθμητικές σταθερές*
- *συναρτήσεις*
- *οδηγία "`include`"*
- *σχόλια*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Το Σύνολο Χαρακτήρων:

Κάθε γλώσσα έχει το αλφάβητό της, δίνοντας έτσι, σε αυτούς που τη χρησιμοποιούν, τη δυνατότητα να τη γράφουν. Ακόμη, στον γραπτό λόγο χρησιμοποιούμε και άλλα σύμβολα όπως αριθμούς, σημεία στίξης κλπ.

Τα παραπάνω ισχύουν και για τις γλώσσες προγραμματισμού. Για να δώσουμε στον ΗΥ ένα πρόγραμμά μας, πρέπει να το κωδικοποιήσουμε, χρησιμοποιώντας το αλφάβητο της γλώσσας και με βάση τους συντακτικούς κανόνες της. Το **σύνολο χαρακτήρων** (character set) της γλώσσας είναι μια διεύρυνση της έννοιας του αλφάβητου και καθορίζει με ακρίβεια το ποιούς χαρακτήρες μπορούμε να χρησιμοποιούμε όταν γράφουμε τα προγράμματά μας. Από την άλλη μεριά, αυτούς τους χαρακτήρες (τουλάχιστον) πρέπει να αναγνωρίζει σωστά ο ΗΥ που θα δώσουμε το πρόγραμμά μας.

Ας δούμε πρώτα μερικούς χαρακτήρες που χρησιμοποιούν όλες οι γλώσσες προγραμματισμού.

Κατ' αρχάς τα **γράμματα** (letters) του *Λατινικού* αλφάβητου, κεφαλαία:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

και πεζά:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Από εδώ και πέρα με το *γράμμα* θα εννοούμε *κεφαλαίο ή μικρό γράμμα του Λατινικού αλφαβήτου*. Όταν θέλουμε να αναφερθούμε σε γράμμα του Ελληνικού αλφαβήτου θα το τονίζουμε.

Πριν αφήσουμε τα γράμματα θα τονίσουμε ότι

- ***Η C++ ξεχωρίζει τα μικρά γράμματα από τα κεφαλαία!***[1]

παρ' όλο που ακόμη δεν μπορείς να καταλάβεις τι ακριβώς σημαίνει αυτό.

Ακόμη, οι γλώσσες χρησιμοποιούν και τα **ψηφία** (digits) του δεκαδικού συστήματος:

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

Πολύ συχνά θα συναντήσεις τον όρο **αλφαριθμητικός χαρακτήρας** (alphanumeric character). Με αυτόν εννοούμε κάποιον χαρακτήρα που είναι είτε γράμμα είτε ψηφίο.

Ένας άλλος χαρακτήρας που χρειάζονται οι γλώσσες προγραμματισμού είναι το **διάστημα** (space) ή **κενό** (blank). Φυσικά, το κενό είναι ένα μέρος του κειμένου όπου δεν υπάρχει κανένα γραφικό σύμβολο. Παρ' όλα αυτά μερικές φορές θα θέλουμε να τονίσουμε την ύπαρξή του. Τότε θα χρησιμοποιούμε το σύμβολο "␣". Αλλού θα δεις για την ίδια χρήση το σύμβολο "**ƀ**".

Παράδειγμα

```
TPITH␣1␣IAN␣2013
TPITHƀ1ƀIANƀ2013
```

Εκτός από τα γράμματα, τα ψηφία και το κενό, οι γλώσσες προγραμματισμού χρησιμοποιούν ορισμένους **ειδικούς χαρακτήρες**, π.χ.:

```
( ) . , + - * / =
```

Κάθε ΗΥ διαθέτει το δικό του σύνολο χαρακτήρων και συνήθως αυτό το σύνολο είναι διαθέσιμο σε αυτόν που γράφει ένα πρόγραμμα σε οποιαδήποτε γλώσσα. Έτσι λοιπόν, μπορεί να έχουμε στην διάθεσή μας περισσότερους χαρακτήρες από αυτούς που είδαμε. Αντίθετα, σε μερικές περιπτώσεις μπορεί να έχουμε περιορισμούς. Για τον λόγο αυτόν, το πρότυπο της κάθε γλώσσας προτείνει πάγιες αντικαταστάσεις αν κάποιοι χαρακτήρες δεν είναι διαθέσιμοι.

---

[1] Οι περισσότερες γλώσσες προγραμματισμού δεν τα ξεχωρίζουν.

1.1 Το Αλφάβητο (Σύνολο Χαρακτήρων) της C++

Αφού είδαμε τους χαρακτήρες που χρησιμοποούν όλες οι γλώσσες προγραμματισμού, ας δούμε τις ιδιαιτερότητες της C++. Όπως καταλαβαίνεις η διαφορά βρίσκεται στους **είδικούς χαρακτήρες**. Η C++ χρησιμοποιεί –περα από αυτούς που χρησιμοποιούν όλες οι γλώσσες– και τους:

```
< > % : ; ? ^ & | ~ ! \ " ' _ { } [ ] #
```

Στη C++, κάθε ειδικός χαρακτήρας παίζει κάποιο ιδιαίτερο ρόλο.

Για υπολογιστές που έχουν «φτωχό» σύνολο χαρακτήρων το πρότυπο της γλώσσας προτείνει πάγιες αντικαταστάσεις για τους χαρακτήρες δεν είναι διαθέσιμοι· αντικαθίστανται από **διγραφικές** (digraph) ή **τριγραφικές** (trigraph) ακολουθίες ή από λέξεις. Για παράδειγμα:

ο `{` αντικαθίσταται από τους `<%` και ο `}` από τους `%>`
ο `[` αντικαθίσταται από τους `<:` και ο `]` από τους `:>`
ο `#` αντικαθίσταται από τους `%:`

Στο Παράρτημα D υπάρχει πίνακας αυτών των αντικαταστάσεων.

1.2 Το Πρώτο Πρόγραμμα

Το πρώτο πρόγραμμα που θα δούμε είναι πολύ απλό και δεν κάνει πολλά πράγματα.

```
#include <iostream>

int main()
{
   std::cout << " Να το πρώτο μου πρόγραμμα" << std::endl;
   std::cout << " ΔΟΥΛΕΥΕΙ!" << std::endl;
   std::cout << " Είμαι στο τρένο της Τεχνολογίας"<< std::endl;
   std::cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << std::endl;
}
```

Γράψε *προσεκτικά* το πρόγραμμα αυτό με τον κειμενογράφο σου, μεταγλώττισέ το και ζήτα από τον ΗΥ σου να το εκτελέσει. Αποτέλεσμα που θα δεις στην οθόνη σου:

```
Να το πρώτο μου πρόγραμμα
ΔΟΥΛΕΥΕΙ!
Είμαι στο τρένο της Τεχνολογίας
ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!
```

Σύγκρινε το αποτέλεσμα με το πρόγραμμα και μάντεψε! Με την εντολή "`std::cout << "`*κείμενο*`"`" ζητάς να γραφεί το κείμενο στην οθόνη του υπολογιστή σου. Με το "`<< std::endl`", στη συνέχεια, ζητάς να τελειώσει η γραμμή και ό,τι γραφεί στη συνέχεια να γραφεί στην επόμενη γραμμή. Τα άλλα τι είναι; Στην επόμενη παράγραφο τα εξηγούμε.

1.2.1 Να «Διώξουμε» το "`std::`"!

Πριν κάνουμε οτιδήποτε άλλο ας δούμε τι είναι αυτό το "`std::`" –που εμφανίζεται κάθε τόσο– και πώς μπορούμε να το διώξουμε.

Από το επόμενο κεφάλαιο θα καταλάβεις ότι τα πρόγράμματά σου θα έχουν πολλά ονόματα, όπως είναι τα *cout* και *endl*. Τα περισσότερα από αυτά θα είναι δικής σου επιλογής. Αργότερα θα δεις ότι για πολλά πράγματα που θέλεις να κάνεις υπάρχουν ήδη λύσεις σε βιβλιοθήκες προγραμμάτων, που φυσικά θα θέλεις να χρησιμοποιήσεις στα προγράμματα που θα γράφεις. Δεν αποκλείεται καθόλου ορισμένα ονόματα να χρησιμοποιούνται στο πρόγραμμά σου και σε κάποια βιβλιοθήκη για να παραστήσουν διαφορετικά αντικείμενα.

Η C++ για να διεκολύνει την κατάσταση δίνει τον εξής μηχανισμό: Μπορείς να δηλώσεις έναν **ονοματοχώρο** (namespace) μέσα στον οποίο δηλώνεις διάφορα ονόματα για

κάποιες χρήσεις. Αν λοιπόν έχεις το όνομα "*nm*" δηλωμένο σε δύο διαφορετικούς ονοματοχώρους *A*, *B* για δύο διαφορετικά προγραμματιστικά αντικείμενα μπορείς να τα ξεχωρίζεις γράφοντας `A::nm` και `B::nm`.

Το προγραμματιστικό περιβάλλον της C++ έχει έναν ονοματοχώρο για τα ονόματα των δικών του αντικειμένων, τον *std* (από το *st*andard). Έτσι, γράφοντας `std::cout` εννοούμε: το «το *cout* που έχει δηλωθεί στον ονοματοχώρο *std*.»

Πρόσεξε τώρα έναν άλλον τρόπο γραφής του προγράμματος:

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

Αυτό το πρόγραμμα δουλεύει μια χαρά! Οι δύο δηλώσεις "`using`" λένε: «χρησιμοποιώ το *std*`::`*cout* ως *cout* και το *std*`::`*endl* ως *endl*».

Υπάρχει και άλλος τρόπος, πιο «τεμπέλικος»:

```
#include <iostream>

using namespace std;

int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

Το "`using namespace std`" σημαίνει ότι χρησιμοποιώ ονόματα από τον *std*. Άρα όλα τα «άγνωστα» ονόματα είναι από εκεί.

Εδώ θα χρησιμοποιούμε τον τελευταίο τρόπο.

Για ονοματοχώρους θα μιλήσουμε εκτενώς αργότερα.

1.3 Πρόγραμμα

Όπως είδες και πιο πάνω, ένα απλό πρόγραμμα C++ είναι μια ακολουθία από εντολές που εκτελούνται η μια μετά την άλλη. Ας δούμε τους συντακτικούς κανόνες της C++ για τη σύνταξη ενός προγράμματος.

Ένα απλό πρόγραμμα αποτελείται από μια *συνάρτηση*, που με τη σειρά της αποτελείται από:

- μια επικεφαλίδα: "`int main()`",
- το *σώμα* της συνάρτησης.

Όπως θα δούμε αργότερα, μέσα στις παρενθέσεις, μπορεί να υπάρχουν οι παράμετροι της συνάρτησης.

Το **σώμα** (body), είναι το μέρος που δίνουμε στον υπολογιστή τις οδηγίες που θέλουμε να εκτελέσει. Προς το παρόν θα έχει μια απλή δομή: θα ξεκινάει με το σύμβολο "`{`" και θα τελειώνει με το σύμβολο "`}`". Ανάμεσα σε αυτά υπάρχουν εντολές που κάθε μια τελειώνει με το χαρακτήρα "`;`".

- ***Από δυο εντολές που γράφονται διαδοχικώς, η εντολή που γράφεται πρώτη θα εκτελεστεί, χρονικώς, πριν από την εντολή που γράφεται δεύτερη.***

Ας δούμε τι αποτέλεσμα θα είχε η αντιμετάθεση των δυο τελευταίων εντολών του προγράμματός μας:

```
#include <iostream>
using namespace std;
int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
}
```

Αποτέλεσμα:

```
 Να το πρώτο μου πρόγραμμα
 ΔΟΥΛΕΥΕΙ!
 ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!
 Είμαι στο τρένο της Τεχνολογίας
```

Ανάμεσα σε δυο εντολές, μπορεί να υπάρχουν οσαδήποτε κενά και οσεσδήποτε αλλαγές γραμμής. Αυτό δεν θα αλλάξει το αποτέλεσμα. Το παράδειγμά μας θα μπορούσε να γραφεί το ίδιο σωστά ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
cout<<" Να το πρώτο μου πρόγραμμα"<<endl;cout<<
" ΔΟΥΛΕΥΕΙ!"<<
endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

ή ως εξής:

```
#include <iostream>
using namespace std;
int main()
{
   cout << " Να το πρώτο μου πρόγραμμα" << endl;

   cout << " ΔΟΥΛΕΥΕΙ!" << endl;

   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;

   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
}
```

Οι πεπειραμένοι προγραμματιστές χρησιμοποιούν τα κενά και τις αλλαγές γραμμής για να κάνουν τα προγράμματά τους πιο *ευανάγνωστα*.

Και εκείνο το

```
#include <iostream>
```

τι είναι; Είναι μια οδηγία προς το μεταγλωττιστή να περιλάβει στο σημείο αυτό ένα αρχείο, με το όνομα `iostream` (μερικά περιβάλλοντα μπορεί να έχουν διαφορετική κατάληξη) που έρχεται μαζί με το μεταγλωττιστή. Αυτό είναι απαραίτητο για να μπορέσει ο μεταγλωττιστής να καταλάβει τα "`cout`", "`<<`", "`endl`". Αν θέλεις να καταλάβεις –κάπως– τι γίνεται, διάβασε την επόμενη παράγραφο· αλλιώς θα τη γράφεις έτσι κι ας μην καταλαβαίνεις τι συμβαίνει.

Τα "`main`", "`cout`" και "`endl`" είναι **ονόματα** (identifiers). Είναι προκαθορισμένα για μερικά χρήσιμα αντικείμενα που χρησιμοποιούμε στα προγράμματά μας. Αργότερα θα δούμε πώς μπορούμε να ορίσουμε δικά μας ονόματα.

1.4 * Η Οδηγία "include"

Ας κάνουμε το εξής πείραμα: Φύλαξε σε ένα αρχείο, με το όνομα **entoles.txt**, τα εξής:

```
   cout << " Να το πρώτο μου πρόγραμμα" << endl;
   cout << " ΔΟΥΛΕΥΕΙ!" << endl;
   cout << " ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!" << endl;
   cout << " Είμαι στο τρένο της Τεχνολογίας" << endl;
```

ενώ στο **prwto.cpp** φύλαξε τα:

```
#include <iostream>
using namespace std;
int main()
{
#include "entoles.txt"
}
```

Τώρα ζήτησε να μεταγλωττισθεί το **prwto.cpp** και να εκτελεσθεί το **prwto.exe**.[2] Αποτέλεσμα; Αυτό που ήδη ξέρεις:

```
Να το πρώτο μου πρόγραμμα
ΔΟΥΛΕΥΕΙ!
Είμαι στο τρένο της Τεχνολογίας
ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!
```

Βγάζεις κάποιο συμπέρασμα για την include; Αποτέλεσμα της

```
#include "entoles.txt"
```

είναι να αντικατασταθεί, πριν από τη μεταγλώττιση, από το περιεχόμενο του αρχείου **entoles.txt**. Έτσι, το πρόγραμμα που πηγαίνει για μεταγλώττιση είναι αυτό που γράψαμε αρχικά, στην §1.2.

Δηλαδή και η "**#include <iostream>**" αντικαθίσταται από κάποιο αρχείο; Ναι, από το αρχείο **iostream**[3]· αν η Dev-C++ και είναι φυλαγμένη στο **c:\Dev-Cpp**, το **iostream** βρίσκεται στο **c:\Dev-Cpp\include\c++\3.4.2**. Μπορείς να το δεις στον κειμενογράφο σου[4]. Στο αρχείο αυτό υπάρχει ο ορισμός του *ρεύματος cout* και του *τελεστή* "**<<**".

Τώρα, θα ρωτήσεις: Γιατί το δικό μας αρχείο το γράψαμε μέσα σε αποστρόφους ενώ το **iostream** το βάζουμε μέσα σε "**<**" και "**>**"; Αυτό καθορίζει τη σειρά με την οποία θα ψάξει ο μεταγλωττιστής διάφορους καταλόγους για να βρει το αρχείο που ζητάμε.

Οι μεταγλωττιστές της C περιλαμβάνουν ένα πρόγραμμα, τον **προεπεξεργαστή** (preprocessor). Ο προεπεξεργαστής, εκτός των άλλων είναι αυτός που εκτελεί τις "**include**" και δημιουργεί το «συμπληρωμένο» αρχείο που θα επεξεργαστεί ο μεταγλωττιστής. Για αυτόν το λόγο η

#include <*όνομα αρχείου***>** ή

#include "*όνομα αρχείου***"**

λέγεται **οδηγία** (directive) **προς τον προεπεξεργαστή**.

Και ποια η διαφορά της πρώτης μορφής από τη δεύτερη;

- Στην πρώτη περίπτωση ο προεπεξεργαστής θα αναζητήσει το αρχείο σε ευρετήρια που είναι προκαθορισμένα για το κάθε περιβάλλον ανάπτυξης. Για παράδειγμα στη Dev-C++, που λέγαμε πιο πάνω, στο **c:\Dev-Cpp\include** και τα υποευρετηριά του.
- Στη δεύτερη περίπτωση η αναζήτηση θα γίνει κατ' αρχάς στο ευρετήριο που βρίσκεται το πρόγραμμά σου. Αν δεν το βρει θα συνεχίσει την αναζήτηση σαν να είχαμε δώσει "**#include <***όνομα αρχείου***>**".

---

[2] Ή όπως αλλιώς ονομάζει το τελικό (εκτελέσιμο) πρόγραμμα το ΛΣ που δουλεύεις.

[3] Μπορεί και **iostream.h** μπορεί και **iostream.hpp**.

[4] Αλλά, για το καλό σου, μην το πειράξεις! Είπαμε να το δεις μόνον!

1.5 Ορμαθοί Χαρακτήρων

Στο πρώτο μας πρόγραμμα, στις εντολές εκτύπωσης, χρησιμοποιήσαμε τα:

```
" Να το πρώτο μου πρόγραμμα"
" ΔΟΥΛΕΥΕΙ!"
" Είμαι στο τρένο της Τεχνολογίας"
" ΖΗΤΩ Η ΠΛΗΡΟΦΟΡΙΚΗ!!!"
```

Αυτά είναι παραδείγματα ορμαθών χαρακτήρων.

Ένας **ορμαθός χαρακτήρων**[5] (string literal) είναι μια ακολουθία χαρακτήρων που περικλείνεται μέσα σε διπλές αποστρόφους (**"**). Οι χαρακτήρες ενός ορμαθού δεν είναι απαραίτητο να ανήκουν στο σύνολο χαρακτήρων της C++. Αρκεί να υπάρχουν στο σύνολο χαρακτήρων του υπολογιστή μας.

Παραδείγματα

```
"α"   "A"   ";"   "α1"   "C++"
"Αυτά είναι νέα"   "12/15*()="    "   "
```

Κατ' αρχήν, *μέσα σ' έναν ορμαθό δεν μπορούμε να έχουμε αλλαγή γραμμής*. Δηλαδή, η αρχική και η τελική διπλή απόστροφος πρέπει να βρίσκονται στην ίδια γραμμή.

Μήκος (length) του ορμαθού είναι το πλήθος των χαρακτήρων που υπάρχουν ανάμεσα στις αποστρόφους.

Προσοχή! Το *διάστημα* (*κενό*) *είναι* χαρακτήρας και το μετράμε στο μήκος.

Οι ορμαθοί που δώσαμε πιο πάνω έχουν μήκη: 1, 1, 1, 2, 3, 14, 9, 3 αντιστοίχως.

Όπως καταλαβαίνεις υπάρχει ένα μικρό πρόβλημα όταν θελήσουμε να περιλάβουμε σε έναν ορμαθό τη διπλή απόστροφο (**"**). Το πρόβλημα λύνεται με την εξής σύμβαση: *Αν θέλουμε να περιλάβουμε μια διπλή απόστροφο σε έναν ορμαθό, τότε πρέπει να την γράψουμε ως* **\"**, που όπως είπαμε ονομάζεται **ακολουθία διαφυγής** (escape sequence). Φυσικά, στον υπολογισμό του μήκους, οι δυο αυτοί χαρακτήρες μετρούνται ως ένας. Με ακολουθίες διαφυγής εισάγονται και οι χαρακτήρες ****, **?**, **'**.

Παραδείγματα

```
"\""  "What\'s up man\?"  "T\' AHTOY TO NYXI"
```

Αυτοί οι ορμαθοί έχουν μήκη: 1, 14, 16 αντιστοίχως.

Στα παραδείγματα που δώσαμε παραπάνω, χρησιμοποιήσαμε και χαρακτήρες που δεν ανήκουν στο σύνολο χαρακτήρων της C++. Στους ΗΥ που έχουμε στην Ελλάδα υπάρχουν και τα ελληνικά γράμματα. Μπορείς να χρησιμοποιείς τους χαρακτήρες αυτούς αλλά να θυμάσαι ότι: *αν μεταφέρεις το πρόγραμμά σου σε άλλον ΗΥ είναι πολύ πιθανό να χρειαστεί να αλλάξεις αυτούς τους χαρακτήρες.*[6]

Ας δούμε ένα άλλο παράδειγμα όπου χρησιμοποιούμε ορμαθούς χαρακτήρων.

Παράδειγμα

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

[5] Μπορεί να το δεις και ως **συμβολοσειρά**.

[6] Μήπως βλέπεις ήδη το πρόβλημα σε διαφορετικά περιβάλλοντα του ίδιου υπολογιστή (αν δουλεύεις σε Windows);!

Αυτό το πρόγραμμα θα δώσει:

```
   *
  * *
 *****
*     *
```

Μπορείς να προκαλέσεις αλλαγή γραμμής όταν γράφεται (π.χ. στην οθόνη) ο ορμαθός, εισάγοντας σε αυτόν την *ακολουθία διαφυγής* "`\n`". Δηλαδή, η

```
   cout << "abc\ndef" << endl;
```

θα γράψει:

```
abc
def
```

1.5.1 Ορμαθοί Μεγάλου Μήκους

Ο περιορισμός «*μέσα σ' έναν ορμαθό δεν μπορούμε να έχουμε αλλαγή γραμμής*» περιορίζει και το μήκος του ορμαθού. Αν τώρα θέλεις να γράψεις έναν πολύ μακρύ ορμαθό έχεις δύο τρόπους:

Ο πρώτος τρόπος είναι να τον σπάσεις σε διαφορετικές γραμμές χρησιμοποιώντας τον χαρακτήρα '\'. Π.χ., η

```
   cout << "abc\
def" << endl;
```

θα γράψει:

```
abcdef
```

Όπως θα δεις και αργότερα, ο χαρακτήρας '\' είναι η λύση που δίνει η C++ όπου υπάρχει περιορισμός μη αλλαγής γραμμής.

Ο δεύτερος τρόπος είναι η πράξη της σύνδεσης: Αν στο πρόγραμμά σου γράψεις δύο ορμαθούς χαρακτήρων στη σειρά, η C++ θα σχηματίσει την **σύνδεσή** (concatenation) τους, που προκύπτει από τους χαρακτήρες του πρώτου ορμαθού ακολουθούμενους από αυτούς του δεύτερου. Π.χ. η εντολή:

```
   cout << "abc" "def" << endl;
```

θα δώσει:

```
abcdef
```

1.6 Έξοδος Αποτελεσμάτων

Η πρώτη εντολή που είδαμε ήταν η εντολή

`cout << "`*κείμενο*`" << endl`

Το αποτέλεσμά της είναι να γραφεί μια γραμμή. Πού; Μάλλον είδες αυτή τη γραμμή στην οθόνη του μικροϋπολογιστή σου (ή του τερματικού σου).

Κάθε ΗΥ έχει μια προκαθορισμένη συσκευή, ή αρχείο, όπου γίνεται η έξοδος των αποτελεσμάτων, εκτός αν ο χρήστης ορίσει κάτι άλλο. Είναι αυτό που τα εγχειρίδια για τον χρήστη αναφέρουν ως *default output device* (*file*). Τα πιο συνηθισμένα παραδείγματα είναι κάποια οθόνη ή ο εκτυπωτής. Αυτή η συσκευή (αρχείο), σε κάθε υλοποίηση της C++, συνδέεται με το πρόγραμμά σου με ένα προκαθορισμένο *ρεύμα* με το όνομα `cout`.

Να δούμε τώρα τη μορφή της εντολής. Η πιο απλή μορφή είναι:

```
cout << endl;
```

Το αποτέλεσμά της είναι να «γραφεί» μια κενή γραμμή.

Γενικώς, μετά τη λέξη *cout* βάζουμε τα *ορίσματά* της, που μεταξύ τους διαχωρίζονται με τους χαρακτήρες "`<<`". Προς το παρόν, τα ορίσματα μπορεί να είναι ορμαθοί χαρακτήρων.

Αποτέλεσμα της εκτέλεσης είναι να γραφεί μια γραμμή που περιέχει τα περιεχόμενα όλων των ορμαθών, με τη σειρά που δίνονται και χωρίς διαχωρισμό μεταξύ τους.

Παράδειγμα

Η εντολή:

```
cout << " ΜΗΛΟ" << "ΠΙΤΑ" << endl;
```

γράφει:

```
 ΜΗΛΟΠΙΤΑ
```

ενώ η εντολή:

```
cout << " AB" << "CD" << "EF" << endl;
```

γράφει:

```
 ABCDEF
```

Και τι θα γίνει αν δεν βάλουμε το `endl`; Τότε, δεν θα έχουμε αλλαγή γραμμής. Π.χ. οι:

```
cout << "abc";
cout << "def" << endl;
```

θα δώσουν:

```
abcdef
```

Αντί για το `endl` μπορεί να δεις να βάζουν το `'\n'`, π.χ.:

```
cout << " AB" << "CD" << "EF" << '\n';
```

Είναι *σχεδόν* το ίδιο πράγμα.

1.7 Αριθμητικές Πραγματικές Σταθερές

Τι είναι μια **σταθερά** (constant) στα μαθηματικά; Είναι μια ποσότητα που δεν αλλάζει η τιμή της. Π.χ.:

```
37,4    -0,1    -1,1    41,00   +10,00017
```

Ας δούμε τις συνιστώσες αυτών των σταθερών και το νόημά τους:

- Στην αρχή μπορεί να έχουμε ένα πρόσημο "`+`" ή "`-`". Αν δεν υπάρχει εννοείται το "`+`".
- Μετά, ακολουθούν τα ψηφία του ακέραιου μέρους. Στα παραδείγματά μας είναι: 37, 0, 1, 41, 10.
- Στη συνέχεια έχουμε την *υποδιαστολή* που διαχωρίζει το ακέραιο από το κλασματικό μέρος. Τη συμβολίζουμε με το "`,`".
- Τέλος υπάρχει το κλασματικό ή δεκαδικό μέρος, που είναι μια ακολουθία ψηφίων.

Οι θετικοί επιστήμονες και οι μηχανικοί σε μερικές περιπτώσεις χρησιμοποιούν την εξής σύμβαση: Αντί να γράψουν

0,000000173 γράφουν $1{,}73\times10^{-7}$

αντί να γράψουν:

3000000000 γράφουν 3×10^{9}

Αυτός ο τρόπος γραφής λέγεται **επιστημονική παράσταση** (scientific notation) ή **παράσταση κινητής υποδιαστολής** (floating point). Σε αντιδιαστολή, η συνηθισμένη γραφή λέγεται **παράσταση σταθερής υποδιαστολής** (fixed point).

Η C++ και οι περισσότερες γλώσσες προγραμματισμού ακολουθούν τους παραπάνω κανόνες με τις εξής διαφορές:

- σημειώνει την υποδιαστολή με τελεία ("`.`") αντί για κόμμα (αμερικανική γραφή αντί της ευρωπαϊκής),

- σημειώνει τη δύναμη του 10 με *eN* αντί για $\times 10^N$. Το *eN*, στην περίπτωση αυτή διαβάζεται –και σημαίνει– «επί 10 στη *N*». Π.χ. το 0.375e3 διαβάζεται «0.375 επί 10 στη 3η» (=375). Ο ακέραιος που ακολουθεί το *e* λέγεται **εκθέτης** (exponent).

Τα παραπάνω παραδείγματα γράφονται στη C++ ως εξής:

Αριθμητική	**C++**
37,4	`37.4`
–0,1	`-0.1`
–1,1	`-1.1`
41,00	`41.00`
10,00017	`10.00017`
$1{,}73\times10^{-7}$	`1.73e-7`
3×10^{9}	`3e9`

Το νόημα των «μεταφρασμένων» σταθερών είναι το ίδιο μ' αυτό που έχουν οι αντίστοιχες αριθμητικές σταθερές.

Τις σταθερές που είδαμε πιο πάνω ονομάζουμε **σταθερές κινητής υποδιαστολής** (floating point constants ή literals) ή *πραγματικές* σταθερές. Δίνουμε πρώτα το συντακτικό τους σε BNF:

σταθερά κινητής υποδιαστολής = *κλασματική σταθερά* [*τμήμα εκθέτη*] |
ακολουθία ψηφίων, *τμήμα εκθέτη*;
κλασματική σταθερά = [*ακολουθία ψηφίων*] "`.`", *ακολουθία ψηφίων* |
ακολουθία ψηφίων, "`.`";
τμήμα εκθέτη = "`e`" [*πρόσημο*] *ακολουθία ψηφίων* |
"`E`" [*πρόσημο*] *ακολουθία ψηφίων*;
πρόσημο = "`+`" | "`-`";
ακολουθία ψηφίων = *ψηφίο* | *ακολουθία ψηφίων*, *ψηφίο*;

Δηλαδή:

- Στην αρχή μπορεί να έχουμε μια κλασματική σταθερά, π.χ.:

```
1234.5678      .1357      2345.
```

Μια κλασματική σταθερά *είναι* σταθερά κινητής υποδιαστολής.

- Η κλασματική σταθερά μπορεί να ακολουθείται από ένα τμήμα εκθέτη που αποτελείται από ένα "`e`" ή "`E`", ένα πρόσημο "`+`" ή "`-`", που μπορεί να παραλείπεται και τέλος μια ακολουθία ψηφίων, π.χ.:

```
1234.5678e+12   1234.5678E-12   1234.5678e12
  .1357e+15       .1357E-15      .1357e15
 2345.e+15       2345.E-15      2345.e15
```

- Στην αρχή μπορεί, αντί για κλασματική σταθερά, να έχουμε μια ακολουθία ψηφίων. Στην περίπτωση αυτή *πρέπει* να υπάρχει εκθέτης:

```
2345e+15       2345E-15       2345e15
```

Και το πρόσημο πριν από τα ψηφία; Σαφέστατα μπορεί να υπάρχει, αλλά η `-1.7e-7` θεωρείται από τη C++ παράσταση.

Εδώ θα πρέπει να προλάβουμε μια πολύ συνηθισμένη παρανόηση:

♦ ***Το "e" δεν σημειώνει πράξη!***

Έτσι, το `5.36e-8` συμβολίζει τον *αριθμό* 0.0000000536 και *όχι την πράξη* $5.36/10^8$.

Τι θα πει αυτό δηλαδή; Αν το "`e`" ήταν πράξη θα μπορούσες να γράψεις

```
5.36e-8e2
```

για να συμβολίσεις την πράξη 0.0000000536 × 100. Αλλά το "`5.36e-8e2`" είναι *συντακτικό λάθος*.

Τέλος, για τις σταθερές κινητής υποδιαστολής ισχύει το εξής:

♦ ***Όταν γράφεις μια πραγματική σταθερά δεν επιτρέπεται να παρεμβάλλεις κενά ή αλλαγές γραμμής.***

Σύμφωνα με τα παραπάνω, τα παρακάτω είναι σταθερές κινητής υποδιαστολής (οι δύο μαζί με ένα πρόσημο):

```
5.67e+4      12.0    56700.0  0.00001  -0.0
    +0.70135e+1  7.0135  .12       19.
```

ενώ τα παρακάτω *δεν είναι:*

`23A`	έχει το γράμμα "`A`"
`3,14`	έχει το "`,`"
`7×10²`	τα "`×`" και "`²`" δεν αναγνωρίζονται
`3 . 14`	έχει κενά
`317.`	έχει αλλαγή γραμμής (και κενά)
`123`	

Μπορείς να ζητήσεις την εκτύπωση μιας αριθμητικής σταθεράς όπως ακριβώς κάνεις και με τους ορμαθούς χαρακτήρων. Τι θα πάρεις όμως; Για δες το

Παράδειγμα

```
#include <iostream>
using namespace std;
int main()
{
   cout << 5.67e+4 << endl;  cout << 12.0 << endl;
   cout << -56700.0 << endl; cout << 0.00001 << endl;
   cout << -0.0 << endl;     cout << +0.70135e+1 <<endl;
   cout << 7.0135 << endl;   cout << .12 << endl;
   cout << 19. << endl;
}
```

Αποτέλεσμα:

```
56700
12
-56700
1e-05
0
7.0135
7.0135
0.12
19
```

Τα αποτελέσματα είναι σωστά, αλλά δεν τυπώθηκαν όπως τα γράψαμε στο πρόγραμμά μας.

Από το τελευταίο παράδειγμα καταλαβαίνουμε ότι:

- Μπορούμε σε μια εντολή εκτύπωσης να βάζουμε ως ορίσματα και αριθμητικές σταθερές.
- Η τιμή της σταθεράς τυπώνεται όχι κατ' ανάγκη όπως τη γράψαμε στο πρόγραμμά μας. Αυτό γίνεται διότι η σταθερά «μεταφράζεται» στην εσωτερική παράσταση που χρησιμοποιεί η C++ για τις σταθερές κινητής υποδιαστολής και στη συνέχεια τυπώνεται με τους κανόνες που γίνεται η εκτύπωση τέτοιων τιμών.

1.7.1 Εσωτερική Παράσταση Πραγματικών Τιμών

Για να καταλάβεις πώς αποθηκεύονται οι τιμές κινητής υποδιαστολής στη μνήμη του ΗΥ, σκέψου ότι γράφονται σε μορφή κινητής υποδιαστολής αλλά στο δυαδικό σύστημα:

$$\sigma\ M \times 2^E$$

όπου το σ είναι πρόσημο και οι M, E δυαδικοί αριθμοί. Στη μνήμη αποθηκεύονται τα:

- σ: σε ένα δυαδικό ψηφίο (0 για το "+", 1 για το "-"),
- M: σε πλήθος δυαδικών ψηφίων που εξαρτάται από την υλοποίηση ή/και τον ΗΥ,
- E: σε πλήθος δυαδικών ψηφίων που εξαρτάται από την υλοποίηση ή/και τον ΗΥ.

Όπως βλέπεις κάθε τιμή κινητής υποδιαστολής πιάνει στη μνήμη τον ίδιο χώρο, που εξαρτάται από τον ΗΥ (ή το λογισμικό του) αλλά όχι από την τιμή που αποθηκεύεται. Αποτέλεσμα; το βλέπεις στο παρακάτω

Παράδειγμα

```
#include <iostream>
using namespace std;
int main()
{
   cout << 12.3456 << endl;
   cout << 12.34567890123456789 << endl;
}
```

Αποτέλεσμα:

```
12.3456
12.3457
```

Τι έγινε εδώ πέρα; Στη δεύτερη σταθερά, εμείς δώσαμε δεκαεννιά ψηφία και μας έβγαλε έξη! Μήπως κάτι δεν πάει καλά με το γράψιμο; Πράγματι, η C++ μας επιτρέπει να καθορίσουμε την **ακρίβεια** (precision) του αποτελέσματος· δες τι μπορούμε να κάνουμε:

```
#include <iostream>
using namespace std;
int main()
{
   cout << 12.3456 << endl;
   cout.precision(20);
   cout << 12.34567890123456789 << endl;
}
```

Όπως θα δούμε και αργότερα, με τη `cout.precision(20)` ζητούμε ακρίβεια είκοσι ψηφίων στα αποτελέσματα που παίρνουμε στο *cout*. Τι θα πάρουμε τώρα;

```
12.3456
12.345678901234567350
```

Μέχρι το 17ο ψηφίο πήγαμε καλά. Από εκεί και πέρα… Εδώ *φτάσαμε στα όρια της ακρίβειας της εσωτερικής παράστασης* και η ακρίβεια στο γράψιμο δεν μπορεί να μας βοηθήσει πια. Έχουμε δηλαδή **απώλεια σημαντικών ψηφίων** (loss of significant digits) κατά την εσωτερική παράσταση της τιμής.

Απώλεια σημαντικών ψηφίων έχουμε και όταν κάνουμε πράξεις με τιμές κινητής υποδιαστολής.

Ο τρόπος εσωτερικής παράστασης βάζει και έναν άλλο περιορισμό: υπάρχει κάποια μέγιστη (και κάποια ελάχιστη) τιμή που μπορεί να παρασταθεί. Αν, κατά κάποιο τρόπο, προκύψει κάποια τιμή έξω από τα προκαθορισμένα όρια έχουμε **υπερχείλιση** (overflow). Τι γίνεται στην περίπτωση αυτή; Εξαρτάται από τη συγκεκριμένη υλοποίηση της C++.

1.8 Ακέραιες Σταθερές

Σε πάρα πολλές περιπτώσεις μπορούμε να δουλέψουμε στα προγράμματά μας με ακέραιες τιμές μόνον. Ας ξεκινήσουμε από τα συντακτικό της **ακέραιης σταθεράς** (integer constant ή literal):

ακέραιη σταθερά = *δεκαδική σταθερά* | "`0`";
δεκαδική σταθερά = *μη-μηδενικό ψηφίο* | *δεκαδική σταθερά*, *ψηφίο*;
μη-μηδενικό ψηφίο = "`1`" | ... | "`9`";

Δηλαδή, μια ακέραιη σταθερά είναι (προς το παρόν) το 0 ή *μια ακολουθία ψηφίων που **δεν** αρχίζει από* 0.

Με το πρόσημο ισχύουν τα ίδια που είπαμε για τις πραγματικές σταθερές, π.χ. το `-375` έχει το νόημα που ξέρουμε, αλλά η C++ το βλέπει ως παράσταση[7].

Παραδείγματα

Τα:

```
12  0  417  375  2048
```

είναι ακέραιες σταθερές. Τα παρακάτω *δεν είναι ακέραιες σταθερές*:

`12.0` έχει την τελεία (“`.`”)
`1E100` έχει το “`E`”
`0,0` έχει το κόμμα (“`,`”)

Οι ακέραιες τιμές αποθηκεύονται με ακρίβεια στη μνήμη του ΗΥ, οι πράξεις τους γίνονται με ακρίβεια και είναι ταχύτερες από αυτές των τιμών κινητής υποδιαστολής. Φυσικά, οι ακέραιες τιμές δεν μπορούν να έχουν κλασματικό μέρος. Ακόμη, τα όριά τους είναι πιο στενά από αυτά που επιτρέπονται για τιμές κινητής υποδιαστολής.

Όπως τυπώνουμε ορμαθούς χαρακτήρων και τιμές κινητής υποδιαστολής μπορούμε να τυπώνουμε και ακέραιες τιμές. Π.χ. η:

```
cout << 12 << endl;
```

δίνει:

```
12
```

1.8.1 Οκταδικοί Ακέραιοι

Γιατί είπαμε «μια ακέραιη σταθερά είναι το 0 ή *μια ακολουθία ψηφίων που **δεν** αρχίζει από* 0»; Τι συμβαίνει με το 0;

Μια ακολουθία ψηφίων που αρχίζει από 0 είναι *οκταδική* (octal) *ακέραιη σταθερά*· έτσι, δεν μπορείς να γράψεις `08` και `09` (θα πάρεις ένα διαγνωστικό σαν “invalid octal constant”) αφού τα "`8`" και "`9`" δεν είναι ψηφία του οκταδικού συστήματος ενώ το `011` σημαίνει 9. Πράγματι, η εντολή

```
cout << 11 << "  " << 011 << endl;
```

θα δώσει:

```
11  9
```

1.8.2 Δεκαεξαδικοί Ακέραιοι

Η C++ σου επιτρέπει να γράφεις ακέραιες σταθερές στο *δεκαεξαδικό* (hexadecimal) σύστημα αρίθμησης. Αρχίζουν με “`0x`” ή “`0X`” και στη συνέχεια έχουν δεκαεξαδικά ψηφία: `0..9` και `a..f` (ή `A..F`). Για παράδειγμα η

```
cout << 0xA << " " << 0xff << " " << 0Xa12C << endl;
```

θα δώσει:

```
10 255 41260
```

1.9 Πράξεις – Αριθμητική Παράσταση

Ένα άλλο είδος ορίσματος για την εντολή εκτύπωσης είναι η **αριθμητική παράσταση** (arithmetic expression). Π.χ. με την εντολή:

```
cout << (1 + 1) << endl;
```

ζητούμε από τον ΗΥ να υπολογίσει το αποτέλεσμα της πράξης `1 + 1` και να το τυπώσει.

---

[7] Το πρόσημο είναι ένας *ενικός* τελεστής, δηλαδή τελεστής που *δρα σε ένα αντικείμενο*.

Πλαίσιο 1.1

Οι Αριθμητικοί Τελεστές της C++

Αριθμητική	C++
+	+
–	-
×	*
/	/
υπόλοιπο:	%

Υπολογισμός Αριθμητικής Παράστασης στη C++

1. Γίνονται οι πράξεις μέσα στις παρενθέσεις και υπολογίζονται οι κλήσεις συναρτήσεων,

2. Μετά γίνονται οι πράξεις *, /, %,

3. Στη συνέχεια γίνονται οι πράξεις +, -.

Οι αριθμητικές πράξεις στη C++ συμβολίζονται όπως και στα μαθηματικά, με μια διαφορά: συμβολίζουμε τον πολλαπλασιασμό με το "`*`" αντί για το "×". Ακόμη, ως σύμβολο της διαίρεσης χρησιμοποιούμε μόνον το "`/`" και όχι το "`:`".

Αν και τα δύο ορίσματα μιας πράξης είναι ακέραιοι το αποτέλεσμα της πράξης είναι ακέραιος, ενώ αν έστω και ένα είναι πραγματικός (π.χ. σταθερά κινητής υποδιαστολής) τότε το αποτέλεσμα είναι πραγματικός[8]. Αυτό, τουλάχιστον προς το παρόν, δεν σημαίνει και πολλά πράγματα εκτός από την περίπτωση της διαίρεσης. Δες το παρακάτω

Παράδειγμα

Το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   cout <<  (1 / 2) << "   " << (1. / 2) << endl;
}
```

δίνει:

```
0   0.5
```

Πώς εξηγείται; Το 0 είναι το αποτέλεσμα της *ακέραιης διαίρεσης* `1 / 2` και, όπως βλέπεις, *δεν* είναι 0.5. Αφού διαιρετέος και διαιρέτης είναι ακέραιοι, ακέραιο είναι και το αποτέλεσμα της διαίρεσης. Η δεύτερη διαίρεση, `1. / 2`, έχει διαιρετέο πραγματικό και έτσι το αποτέλεσμα είναι πραγματικός.

Ειδικά για τη διαίρεση ακεραίων, υπάρχει ένας ακόμη τελεστής ο "`%`" που κάνει μια πολύ γνωστή δουλειά: *μας δίνει το υπόλοιπο της ακέραιης διαίρεσης*. Σχετίζεται με τον "`/`" με τη –γνωστή από την αριθμητική «δοκιμή της διαίρεσης»– σχέση:

$$\Delta = \pi * \delta + \upsilon$$

όπου Δ ο διαιρετέος και δ ο διαιρέτης. Αλλιώς:

$$\Delta = (\Delta \ / \ \delta) * \delta + (\Delta \ \% \ \delta) \qquad (1)$$

Έτσι,

η πράξη: `5 / 2` δίνει 2

η πράξη: `5 % 2` δίνει 1

[8] Στο επόμενο κεφάλαιο θα δεις ότι ο κανόνας είναι πιο πολύπλοκος.

Όπως είναι φυσικό, είναι λάθος να βάλεις δεύτερο όρισμα σε οποιαδήποτε από τις δύο πράξεις το 0 (μηδέν).

Αν κάποιο από τα ορίσματα είναι αρνητικό τότε το αποτέλεσμα της "**%**" είναι τέτοιο ώστε να ισχύει η (1)· δηλαδή *μπορεί να είναι και αρνητικό*! Π.χ. το:

```
#include <iostream>
using namespace std;
int main()
{
   cout <<  (1 / 2) << "   " << (1 % 2) << "   "
        << ((1/2)*2 + (1%2)) << endl;
   cout <<  (1 / (-2)) << "   " << (1 % (-2)) << "   "
        << ((1/(-2))*2 + (1%(-2))) << endl;
   cout <<  ((-1) / 2) << "   " << ((-1) % 2) << "   "
        << (((-1)/2)*2 + ((-1)%2)) << endl;
   cout <<  ((-1) / (-2)) << "   " << ((-1) % (-2)) << "   "
        << (((-1)/(-2))*(-2) + ((-1)%(-2))) << endl;
}
```

θα δώσει:

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

Ήδη στο παραπάνω παράδειγμα βλέπουμε παραστάσεις με περισσότερες από μια πράξεις και με παρενθέσεις. Ας δούμε τι γίνεται σε τέτοιες περιπτώσεις. Π.χ. ας πούμε ότι έχουμε:

```
7*5 - 31.0/2 + 9*2/5.0
```

Πώς θα υπολόγιζες το αποτέλεσμα στην αριθμητική; Πρώτα θα υπολογίσεις τις τιμές των όρων:

```
7*5 = 35
31.0/2 = 15.5
9*2/5.0 = 3.6
```

Δηλαδή πρώτα θα κάνεις πολλαπλασιασμούς και διαιρέσεις. Στη συνέχεια θα κάνεις προσθέσεις και αφαιρέσεις:

```
35 - 15.5 + 3.6 = 23.1
```

Το ίδιο γίνεται και στη C++:

- πρώτα γίνονται οι πράξεις *****, **/**, **%**,
- στη συνέχεια γίνονται οι πράξεις **+**, **-**.

Ωραία, αλλά στην αριθμητική μπορούμε να παραβιάσουμε αυτή τη σειρά των πράξεων με τις παρενθέσεις, με αγκύλες, με άγκιστρα. Εδώ τι γίνεται; Το ίδιο πράγμα, αλλά χρησιμοποιούμε μόνον παρενθέσεις.

♦ ***Ό,τι υπάρχει μέσα στις παρενθέσεις υπολογίζεται πρώτο.***

Η παράσταση:

$$1 + \{ 1 + 4 \times [6 + 2 \times (9 - 3.5) / 2.2] \}$$

θα γραφεί στη C++:

```
1 + ( 1 + 4 * ( 6 + 2 * ( 9 - 3.5 ) / 2.2 ) )
```

Η διαφορά είναι αμελητέα. Μόνο πρόσεχε!

♦ ***Όσες παρενθέσεις ανοίγεις τόσες ακριβώς πρέπει να κλείνεις και –προ πάντων– στο σωστό σημείο.***

Με βάση τους κανόνες που είπαμε μέχρι τώρα, μπορείς να γράψεις σωστά στη C++ σχεδόν οποιαδήποτε αριθμητική παράσταση. *Όταν έχεις κάποια αμφιβολία για την σειρά εκτέλεσης των πράξεων, χρησιμοποίησε παρενθέσεις*, αφού ό,τι βρίσκεται μέσα σ' αυτές υπολογίζεται πρώτο.

Παράδειγμα

Έστω ότι θέλουμε να υπολογίσουμε το:

$$\frac{3.45}{5\times 3.14159}$$

Αν γράψουμε:

```
3.45 / 5 * 3.14159
```

κάνουμε *λάθος*. Όπως είπαμε πιο πριν, οι "`*`" και "`/`" έχουν την ίδια προτεραιότητα. Αν οι πράξεις γίνονται από αριστερά προς τα δεξιά, πρώτα θα υπολογιστεί το: 3.45/5 και ότι βρεθεί θα *πολλαπλασιαστεί* με το 3.14159. Δηλαδή, θα υπολογιστεί το:

$$\frac{3.45}{5}\times 3.14159$$

Χρησιμοποιώντας παρενθέσεις γράφουμε το σωστό:

```
3.45/(5 * 3.14159)
```

Τώρα, θα υπολογιστεί η τιμή της παράστασης μέσα στις παρενθέσεις και μετά θα διαιρεθεί το 3.45 με αυτό που βρήκαμε από τον πολλαπλασιασμό.

1.10 Οι Συναρτήσεις της C++

Στην προηγούμενη παράγραφο λέγαμε ότι τώρα μπορείς να μεταγράψεις σχεδόν κάθε μαθηματική παράσταση σε C++. Καλά που βάλαμε το «σχεδόν», γιατί εσύ θέλεις να μεταγράψεις τη:

$$\frac{\text{συν}(72^\circ)}{1+\sqrt{5}} \qquad \text{και τη} \qquad 2.5+\sqrt{5^2-3^2}$$

Κανένα πρόβλημα! Νάτες:

```
cos(72 * 3.14159 / 180) / (1 + sqrt(5))
2.5 + sqrt(5*5 - 3*3)
```

Τα καινούρια πράγματα εδώ είναι τα εξής: `cos(3.14159 * 72 / 180)`, `sqrt(5)`, `sqrt(5*5 - 3*3)`. Πρόκειται για κλήσεις δυο συναρτήσεων της C++ με τα ονόματα **`cos`**, **`sqrt`**, που μας δίνουν το συνημίτονο και την τετραγωνική ρίζα αντίστοιχα. Εντάξει! Αρχικά όμως είχαμε *συν*(72°). Πώς εμφανίστηκαν αυτά τα 3.14159/180; Λοιπόν: η συνάρτηση *cos()* της C++ περιμένει το όρισμά της σε ακτίνια. Το $\pi/180$ είναι ο παράγοντας μετατροπής.

Κάθε συνάρτηση της C++ έχει ένα όνομα με το οποίο τη χρησιμοποιούμε. Πού; Μέσα σε αριθμητικές παραστάσεις. Γράφουμε το όνομα της συνάρτησης που θέλουμε και στη συνέχεια μέσα σε παρενθέσεις το όρισμα ή τα ορίσματά της. Όλο αυτό είναι μια **κλήση συνάρτησης** (function call).

- ***Σε μια παράσταση, η κλήση συνάρτησης έχει την ίδια προτεραιότητα με μια παράσταση μέσα σε παρενθέσεις, δηλαδή υπολογίζεται στην αρχή, πριν από τις πράξεις.***

  Ο υπολογισμός γίνεται ως εξής:

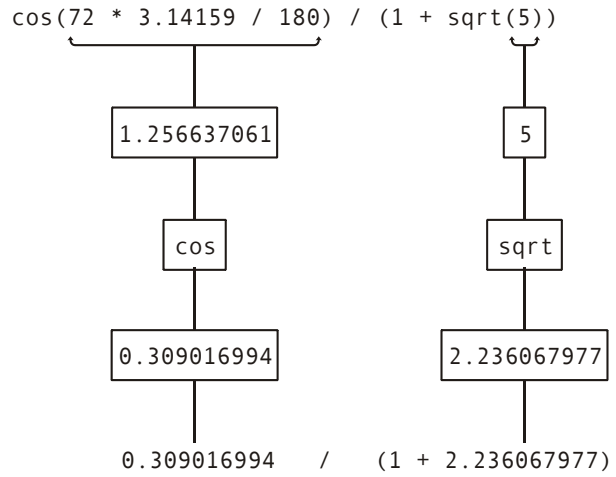
- Πρώτα υπολογίζεται η τιμή του ορίσματος,
- στη συνέχεια η τιμή της συνάρτησης και
- τέλος η τιμή αντικαθίσταται στη θέση της κλήσης συνάρτησης για να γίνουν οι άλλες πράξεις.

Στο Σχ. 1-1 βλέπεις πώς γίνονται τα παραπάνω για το πρώτο παράδειγμα.

Στο Παρ. Β βλέπεις όλες τις **συναρτήσεις** (functions) που υπάρχουν στη μαθηματική βιβλιοθήκη (math) της C++. Για να τις χρησιμοποιήσεις θα πρέπει στην αρχή του προγράμματός σου να βάλεις την οδηγία

```
#include <cmath>
```

Ας δούμε ένα παράδειγμα με μερικές από αυτές:



Σχ. 1-1 **Σειρά των πράξεων για τον υπολογισμό συνάρτησης.**

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   cout << sqrt( 2.0 ) << "    " << sqrt( 4.0*9 )<<endl;
   cout << exp( 1.0 ) << endl;
   cout << log( 1.0 ) <<"    "<<log( exp( 1.0 ) )<<endl;
   cout << sin( 3.141592653 / 3 ) << endl;
   cout << cos( 3.141592653 / 6 ) << endl;
   cout << atan( 1.0 ) << "  π = "<<4*atan( 1.0 )<<endl;
   cout << pow( 10.0,3.0 )<<"  "<<pow( 2.0, 0.5 )<<endl;
   cout << fabs( -7.0 ) <<  "    " << abs( 7 ) << endl;
}
```

1. ***sqrt:*** Μας δίνει την **τετραγωνική ρίζα** (**sq**uare **r**oot) του ορίσματος. Το αποτέλεσμά της είναι τιμή κινητής υποδιαστολής. Η παράσταση που θα μπει ως όρισμα, θα πρέπει να παίρνει τιμή *μη-αρνητική*. Η:

```
   cout << sqrt( 2.0 ) << "    " << sqrt( 4.0*9 )<<endl;
```

θα δώσει:

```
1.41421    6
```

2. ***exp:*** Το *exp*(*x*) μας δίνει το e^x, όπου *e* η βάση των φυσικών λογαρίθμων. Μπορείς να πάρεις από τον ΗΥ σου την τιμή του *e* με την:

```
   cout << exp( 1.0 ) << endl;
```

που θα σου δώσει:

```
2.71828
```

3. ***log:*** Μας δίνει τον **φυσικό λογάριθμο** του ορίσματος. Το όρισμά της πρέπει να έχει *θετική* τιμή. Είναι η αντίστροφη συνάρτηση της *exp()*. Π.χ. η:

```
   cout << log( 1.0 ) << "    " << log( exp(1.0) ) << endl;
```

θα σου δώσει:

```
0    1
```

4. ***sin:*** Μας δίνει το **ημίτονο** του ορίσματος (γωνίας). Το όρισμα θα πρέπει να είναι γωνία (τόξο) σε *ακτίνια*. Π.χ. η παρακάτω εντολή μας τυπώνει το ημίτονο των 60° (= *π*/3):

```
   cout << sin( 3.141592653 / 3 ) << endl;
```

αποτέλεσμα:

```
0.866025
```

5. *cos:* Μας δίνει το **συνημίτονο** του ορίσματος (γωνίας). Το όρισμα θα πρέπει να είναι γωνία (τόξο) σε *ακτίνια*. Π.χ. η παρακάτω εντολή μας τυπώνει το συνημίτονο των 30° (=π/6):

```
   cout << cos( 3.141592653 / 6 ) << endl;
```

αποτέλεσμα:

```
0.866025
```

6. *atan:* Μας δίνει το **τόξο εφαπτομένης** του ορίσματος. Δηλαδή, το τόξο –γωνία– σε ακτίνια, στο διάστημα (–π/2, π/2), που η εφαπτομένη του είναι ίση με το όρισμα.

Αν πάρεις υπόψη σου ότι $\varepsilon\phi(\pi/4) = 1$ και επομένως: $\pi = 4\tau o\xi\varepsilon\phi(1)$, μπορείς να δεις την τιμή του π, όπως την έχει η C++ που δουλεύεις. Η:

```
   cout << atan( 1.0 ) << "  π = " << 4*atan( 1.0 ) << endl;
```

θα δώσει:

```
0.785398  π = 3.14159
```

7. *pow:* Η *pow*(x, y) μας δίνει το x^y. Π.χ.

```
   cout << pow( 10. 0,3.0 ) << "  " << pow( 2.0, 0.5 ) <<endl;
```

αποτέλεσμα:

```
1000    1.41421
```

8. *abs, fabs:* Μας δίνουν την **απόλυτη τιμή** (**abs**olute value) του ορίσματος. Η *abs()* περιμένει ακέραιο όρισμα, ενώ η *fabs()* πραγματικό. Π.χ.

```
   cout << fabs(-7.0) <<  "    " << abs(7) << endl;
```

αποτέλεσμα:

```
7    7
```

1.10.1 "cmath" ή "math.h" ;

Σε πολλά μέρη είδες να γράφεται αντί για "`#include <cmath>`"

```
#include <math.h>
```

Ποιο είναι το σωστό; Το σωστό είναι `cmath` αλλά και το `math.h` θα δουλέψει.

Το `math` είναι κληρονομιά από τη C (ενώ το `iostream`, που χρησιμοποιούμε ήδη, είναι της C++). Ο κανόνας είναι ως εξής:

- Τα περιλαμβανόμενα αρχεία της C++ γράφονται χωρίς (οποιαδήποτε) κατάληξη, π.χ. γράφουμε:

```
#include <iostream>
```

- Τα περιλαμβανόμενα αρχεία της C γράφονται με πρόθεμα "`c`" (και χωρίς κατάληξη), π.χ.:

```
#include <cmath>
```

  Στην περίπτωση αυτή όλες οι δηλώσεις γίνονται στον ονοματοχώρο *std*, π.χ. στην πραγματικότητα έχουμε *std*`::`*fabs*, αλλά δεν έχουμε πρόβλημα αφού έχουμε βάλει το "`using namespace std`".

- Τα περιλαμβανόμενα αρχεία της C γράφονται με κατάληξη "`.h`" (χωρίς πρόθεμα), π.χ.:

```
#include <math.h>
```

  Αυτή η δυνατότητα είναι προσωρινή, αφού υπάρχουν ήδη πολλά προγράμματα που χρησιμοποιούν αυτή τη σύμβαση. Πάντως το πρότυπο της γλώσσας δίνει οδηγία να μην χρησιμοποιείται αυτό ο τρόπος.

1.11 Έλεγχος Εκτύπωσης

Ας δούμε τώρα πώς μπορούμε να ρυθμίσουμε την εμφάνιση των αποτελεσμάτων, ώστε να μην τα γράφει η C++ όπως θέλει. Προς το παρόν θα δούμε μερικά σχετικά εργαλεία· αργότερα θα μάθουμε κι άλλα.

Για κάθε τιμή που γράφεις στο *cout* μπορείς να δώσεις και ένα **πλάτος πεδίου** (field width). Δίνοντας:

```
cout.width(w);
```

όπου `w` ακέραιη τιμή > 0, ζητάς η επόμενη τιμή (αριθμητική ή ορμαθός χαρακτήρων) που θα γραφεί στο *cout να καταλάβει τουλάχιστον* `w` *θέσεις* σε μια γραμμή εκτύπωσης. Π.χ. οι εντολές:

```
cout.width( 4 );   cout << "abcdefghij";
cout.width( 4 );   cout << "ab";
cout.width( 6 );   cout << 1.1;
cout.width( 8 );   cout << 11;
cout.width( 10 );  cout << 0.0001 << endl;
```

θα δώσουν:

```
abcdefghij␣␣ab␣␣␣1.1␣␣␣␣␣␣11␣␣␣␣0.0001
```

και ας δούμε γιατί. Ζητούμε πλάτος πεδίου

- τουλάχιστον 4 για τον ορμαθό `"abcdefghij"`, που έχει μήκος 10. Γράφεται σε 10 θέσεις.
- 4 για τον ορμαθό `"ab"`, που έχει μήκος 2. Γράφεται σε 4 θέσεις, από τις οποίες οι δύο πρώτες (στα αριστερά) έχουν κενά.
- 6 για την πραγματική τιμή `1.1`, που χρειάζεται 3 θέσεις. Γράφεται σε 6 θέσεις, από τις οποίες οι 3 πρώτες (στα αριστερά) έχουν κενά.
- 8 για την ακέραιη τιμή `11`, που χρειάζεται 2 θέσεις. Γράφεται σε 8 θέσεις, από τις οποίες οι 6 πρώτες έχουν κενά.
- 10 για την ακέραιη τιμή `0.0001`, που χρειάζεται 6 θέσεις. Γράφεται σε 10 θέσεις, από τις οποίες οι 4 πρώτες έχουν κενά.

Για τις πραγματικές τιμές, αυτό που μας ενδιαφέρει συνήθως είναι η **ακρίβεια** (precision). Αν δώσεις την εντολή:

```
cout.precision( d );
```

όπου `d` θετική ακέραιη τιμή, ζητάς οι επόμενες πραγματικές τιμές να τυπώνονται με *d* το πολύ σημαντικά ψηφία. Π.χ. οι

```
cout.precision( 8 );
cout << 1234.5 << "  " << 123456.78 << "  "
     << 123456789.0123 << endl;
```

θα δώσουν:

```
1234.5  123456.78  1.2345679e+08
```

Όπως βλέπεις, όλες οι τιμές γράφονται με 8 το πολύ σημαντικά ψηφία.

Ο καθορισμός ακρίβειας αναιρείται με νέα `cout.precision`. Η τιμή που καθορίζεται ερήμην είναι: 6.

Κάτι που ενοχλεί συχνά είναι ότι η C++ αποφασίζει με δικά της κριτήρια για το αν, κάποια τιμή, θα γράφεται σε μορφή σταθερής ή κινητής υποδιαστολής. Σου δίνει όμως τη δυνατότητα να το ελέγξεις. Αν δώσεις:

```
cout.setf( ios::scientific, ios::floatfield );
```

όλες οι πραγματικές τιμές που θα γράφεις στη συνέχεια (και μέχρι να δώσεις μια νέα `cout.setf`) θα γράφονται σε παράσταση κινητής υποδιαστολής. Π.χ. οι:

```
cout.precision( 8 );
cout << 1234.5 << "  " << 123456.78 << "  "
     << 123456789.0123 << endl;
```

θα δώσουν:

```
1.23450000e+03  1.23456780e+05  1.23456789e+08
```

Πρόσεξε τώρα κάτι σε σχέση με την ακρίβεια: Αν έχουμε ακρίβεια *d* ψηφίων για κάθε τιμή θα τυπώνεται:

- το πρόσημο αν είναι "`-`",
- ένα ψηφίο πριν από την υποδιαστολή,
- *d* ψηφία μετά την υποδιαστολή,
- `e±99` ή `e±999` σε τέσσερις ή πέντε θέσεις.

Όπως καταλαβαίνεις, αν έχεις ορίσει ακρίβεια *d* ψηφίων, σου χρειάζονται από *d* + 6 μέχρι *d* + 8 θέσεις. Αν λοιπόν θέλεις να καθορίσεις και το πλάτος πεδίου θα πρέπει να φροντίσεις να είναι μεγαλύτερο.

Αν τώρα δώσεις:

```
cout.setf( ios::fixed, ios::floatfield );
```

οι εντολές:

```
cout.precision( 8 );
cout << 1234.5 << "  " << 123456.78 << "  "
     << 123456789.0123 << endl;
```

θα δώσουν:

```
1234.50000000  123456.78000000  123456789.01230000
```

Όπως βλέπεις, και στην περίπτωση αυτή, η ακρίβεια (εδώ 8) καθορίζει το πλήθος των ψηφίων που θα γραφούν μετά την υποδιαστολή.

Αν θέλεις να επιστρέψεις στον αυτόματο καθορισμό παράστασης δώσε:

```
cout.setf( 0, ios::floatfield );
```

1.12 Κληρονομιά από τη C: *printf()*

Διαβάζοντας άλλα βιβλία (κυρίως για C αλλά μερικές φορές και για C++) θα δεις να χρησιμοποιείται για την έξοδο αποτελεσμάτων η *printf()*. Θα πούμε μερικά πράγματα για να καταλαβαίνεις αυτά που μπορεί να δεις.

Για να χρησιμοποιήσεις σε ένα πρόγραμμα την *printf()* θα πρέπει να περιλάβεις στο πρόγραμμά σου την `cstdio` αντί για την `iostream`.

Ας ξεκινήσουμε με ένα παράδειγμα χρήσης της *printf*:

```
#include <cstdio>
#include <cmath>

int main()
{
   printf( "%s%f) = %f\n", "τετραγωνική ρίζα(",
           2.0, sqrt(2.0) );
}
```

που δίνει:

```
τετραγωνική ρίζα(2.000000) = 1.414214
```

Όπως η C++ έχει το ρεύμα *cout* προς την οθόνη, η C έχει το ρεύμα "`stdout`". και η *printf()* γράφει σε αυτό. Σε γενικές γραμμές: *ό,τι κάνει ο "`<<`" προς το cout κάνει η printf() προς το stdout*.

Όπως βλέπεις το πρώτο όρισμα της *printf()* είναι ένας ορμαθός χαρακτήρων με «περίεργο περιεχόμενο»· είναι ο **ορμαθός μορφοποίησης** (format string) της *printf*. Στον ορμαθό μορφοποίησης υπάρχουν οι **προδιαγραφές μορφοποίησης** (format specifiers) που αρχίζουν πάντοτε με τον χαρακτήρα "`%`". Ας δούμε τι υπάρχει στο παράδειγμά μας.

- Το "`%s`" λέει: «γράψε όπως ξέρεις έναν ορμαθό χαρακτήρων που θα βρεις στη συνέχεια» (πρόκειται για τον: "`τετραγωνική ρίζα(`").
- Το "`%f`" λέει: «γράψε όπως ξέρεις μια πραγματική τιμή που θα βρεις στη συνέχεια σε μορφή σταθερής υποδιαστολής» (πρόκειται για την: `2.0`).
- Στη συνέχεια θα πρέπει να τυπωθούν όπως είναι τα "`)␣=␣`".
- Το "`%f`" λέει και πάλι: «γράψε όπως ξέρεις μια πραγματική τιμή που θα βρεις στη συνέχεια σε μορφή σταθερής υποδιαστολής» (πρόκειται για την: `sqrt(2.0)`).
- Τέλος, το "`\n`" λέει: «αφού γράψεις όλα τα προηγούμενα, άλλαξε γραμμή».

Γενικώς, για πραγματικές τιμές χρησιμοποιούμε την προδιαγραφή `%f` (σταθερή υποδιαστολή) ή `%e` (κινητή υποδιαστολή), για ακέραιη τιμή προδιαγραφή `%d`, για ορμαθούς χαρακτήρων προδιαγραφή `%s` και για έναν χαρακτήρα μόνον προδιαγραφή `%c` (αλλά και `%d`).

Μετά το "`%`" μπορείς να βάλεις έναν φυσικό αριθμό που καθορίζει το πλάτος πεδίου. Στις προδιαγραφές `%f` και `%e` ένας δεύτερος φυσικός καθορίζει τα ψηφία μετά την υποδιαστολή. Η:

```
printf( "%4s%4s%6.1f%8d%10.4f\n",
        "abcdefghij", "ab", 1.1, 11, 0.0001 );
```

θα δώσει:

```
abcdefghij␣␣ab␣␣␣1.1␣␣␣␣␣␣11␣␣␣␣0.0001
```

ενώ η:

```
printf( "%6.1e  %10.4e\n", 1.1, 0.0001 );
```

δίνει:

```
1.1e+00  1.0000e-04
```

Καλύτερα να χρησιμοποιείς τον μηχανισμό εξόδου της C++, αυτόν με το ρεύμα *cout*. Πάντως απόφυγε να χρησιμοποιείς και τους δύο μηχανισμούς στο ίδιο πρόγραμμα.

1.13 Σχόλια

Η C++ μας επιτρέπει να βάζουμε **σχόλια** (comments) μέσα στο πρόγραμμα που να δίνουν σ' αυτόν που το διαβάζει, διάφορες εξηγήσεις. Τα σχόλια υπάρχουν μόνο στο αρχικό πρόγραμμα και αγνοούνται από το μεταγλωττιστή όταν μεταγλωττίζει το πρόγραμμα.

Για την C++, σχόλιο είναι οτιδήποτε υπάρχει:

- σε μια γραμμή προγράμματος μετά το σύμβολο "`//`", π.χ:

```
cout << "1+1 = " << (1+1) << endl; // τα σχόλια περιττεύουν
// αυτό είναι άλλο ένα σχόλιο
```

- ανάμεσα στα σύμβολα "`/*`" και "`*/`". Π.χ.

```
/* ΚΙ ΑΥΤΟ ΕΙΝΑΙ ΕΝΑ ΣΧΟΛΙΟ */
cout << 3.14159 / 2 /* = π/2 */ << endl;
```

Όπως βλέπεις στο τελευταίο παράδειγμα μπορείς να βάζεις τα σχόλια και μέσα στις εντολές που δίνεις. Αυτό πάντως δεν είναι πάντοτε η καλύτερη ιδέα!

Τα σχόλια που θα βάζεις στα προγράμματά σου θα πρέπει να είναι προσεγμένα, ώστε να εξηγούν ό,τι χρειάζεται εξήγηση και να μη κουράζουν με ανοησίες για προφανή πράγματα.

1.14 Προβλήματα;

Σε βασανίζει ο μεταγλωττιστής όταν γράφεις τα προγράμματά σου; Για να δούμε τι προβλήματα μπορεί να έχεις.

Έστω ότι θέλεις να γράψεις ένα πρόγραμμα που να δίνει την τιμή της παράστασης: $1+\sqrt{2}$. Γράφεις:

```
int main()
{
   cout << (1 + sqrt(2)) << endl
}
```

Καλό δεν είναι; Καλό... Το δίνουμε για μεταγλώττιση και... 1ο λάθος:

```
Undefined symbol 'cout' in function int main()
```

Δηλαδή χρησιμοποιούμε το σύμβολο (όνομα) *cout* χωρίς να το έχουμε ορίσει! Το λάθος μας είναι ότι ξεχάσαμε να βάλουμε τα:

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

Το διορθώνουμε και ξαναπροσπαθούμε. 2ο λάθος:

```
Function 'sqrt' should have a prototype in function int main()
```

Αυτή τη φορά το λάθος μας είναι ότι ξεχάσαμε να βάλουμε την οδηγία:

```
#include <cmath>
```

Το διορθώνουμε και προσπαθούμε για τρίτη φορά. 3ο λάθος:

```
Statement missing ; in function int main()
```

Ξεχάσαμε να βάλουμε "`;`" μετά το `endl`!

Το διορθώνουμε και όλα πάνε καλά. Να το πρόγραμμα:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   cout << (1 + sqrt(2)) << endl;
}
```

Στα επόμενα κεφάλαια θα καταλάβεις τι σημαίνουν όσα είπαμε μέχρι τώρα για τις οδηγίες `include`. Προς το παρόν θα πρέπει να κάνεις τα εξής:

- Αν ένα πρόγραμμά σου έχει εντολές `cout << ...` θα πρέπει να έχει στην αρχή την οδηγία `#include <iostream>`.
- Αν ένα πρόγραμμά σου χρησιμοποιεί οποιαδήποτε από τις αριθμητικές συναρτήσεις του Παραρτ. Β θα πρέπει να έχει στην αρχή την οδηγία `#include <cmath>`.

Ας δούμε τώρα ένα λάθος που κάνουν οι πρωτάρηδες. Για το παραπάνω πρόβλημα γράφεις:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   1 + sqrt(2);
}
```

Δίνεις το πρόγραμμα στο μεταγλωττιστή, όλα πάνε καλά, αλλά κανένα αποτέλεσμα· η οθόνη κενή! Τι γίνεται εδώ; Ζητήσαμε να υπολογιστεί η τιμή της παράστασης αλλά *δεν ζητήσαμε να γραφεί*! Θα έπρεπε να είχαμε γράψει:

```
cout << (1 + sqrt(2))...
```

Ασκήσεις

Α Ομάδα

1-1 Ποια από τα παρακάτω είναι πραγματικές σταθερές:

```
1,0
56.000
-234a
89.5
1e1
+1.01e+01
5.e-1
```

1-2 Ποια από τα παρακάτω είναι ακέραιες σταθερές:

```
1.0
1
-1
-1,0
78
999999
67.89
-8989
```

1-3 Ξεχώρισε από τις παρακάτω παραστάσεις, όσες είναι δεκτές ως προς το συντακτικό από τη C++.

Υπολόγισε τις τιμές τους.

```
7*12 + 3
7.*12 + 3
7.0*12 + 3
21 / 5 % 2
12 / 8
1. + 7/3
1.0 - 7 / 3
13 + 4A -12
58E14
7/(12 - 11E-3/7)
17 % 6/3
```

Β Ομάδα

1-4 Γράψε πρόγραμμα που θα υπολογίζει και θα τυπώνει τα: π, π^2, $\sqrt{\pi}$, e, e^2, $\sqrt{e}$ (e: η βάση των φυσικών λογαρίθμων).

1-5 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει τις τιμές των παρακάτω παραστάσεων:

$$\frac{1}{1+\sqrt{5}} \qquad 1+e^{1/2} \qquad \eta\mu(60^o)-\sqrt{2}$$

(e: η βάση των φυσικών λογαρίθμων).

1-6 Γράψε πρόγραμμα C++ που θα υπολογίζει και θα τυπώνει τις τιμές των παραστάσεων:

$$\frac{\sqrt{2+\sqrt{3}}}{2} \qquad ln\left|\frac{1}{1-\sqrt{2}}\right| ln\left|\frac{1}{1+\sqrt{2}}\right| \qquad \frac{\sqrt{2}(\sqrt{3}+1)}{4}$$

($ln(x)$ ο φυσικός (νεπέρειος) λογάριθμος του x).

1-7 Γράψε προγράμματα που υπολογίζουν και τυπώνουν (μαζί με κατάλληλα σχόλια) τα ακόλουθα:

α) Το εμβαδό τριγώνου με βάση 10.5 *cm* και αντίστοιχο ύψος 8.7 *cm*.
β) Τον όγκο κυλίνδρου με περιφέρεια 9.2 *cm* και ύψος 5.3 *cm*.
γ) Την απόσταση δύο σημείων του επιπέδου με συντεταγμένες: (3,2) και (5,8).
δ) Την τιμή του πολυωνύμου x^2-2x+3 για $x = 1,2,3$

1-8 Γράψε πρόγραμμα που θα «ζωγραφίζει» τα αρχικά σου. Τα γράμματα θα πιάνουν πέντε γραμμές και πέντε στήλες το καθένα και θα σχηματίζονται από τον χαρακτήρα "*****" (δες το παράδειγμα της §1.6).

1-9 Στην §1.9 είπαμε ότι: «Αν και τα δύο ορίσματα μιας πράξης είναι ακέραιοι το αποτέλεσμα της πράξης είναι ακέραιος, ενώ αν έστω και ένα είναι πραγματικός τότε το αποτέλεσμα είναι πραγματικός.» Γράφουμε λοιπόν:

```
cout << (4./2) << "  " << (4/2) << endl;
```

και παίρνουμε αποτέλεσμα:

```
2  2
```

Και πού ξέρουμε ότι η πρώτη είναι πραγματική και η δεύτερη ακέραιη; Μπορείς να βρεις έναν τρόπο να πεισθούμε;
Υπόδ.: Δες αυτά που λέμε στην §1.11.