κεφάλαιο

3

* Το Σωστό Πρόγραμμα

Ο στόχος μας σε αυτό το κεφάλαιο:
Να κατανοήσουμε την έννοια της επαλήθευσης προγράμματος και τις βασικές σχετικές διαδικασίες.
Προσδοκώμενα αποτελέσματα:
Να μπορείς να αποδείξεις την ορθότητα απλών προγραμμάτων.
Έννοιες κλειδιά:

- *προδιαγραφές προγράμματος*
- *απόδειξη ορθότητας ή επαλήθευση προγράμματος*
- *συνθήκες επαλήθευσης*
- *αξίωμα της εκχώρησης*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Γράφαμε στην §0.4:

- ***το πρόγραμμα είναι ένα προϊόν για το οποίο πρέπει να υπάρχουν προδιαγραφές,***
- ***το πρόγραμμα γράφεται έτσι ώστε να συμμορφώνεται με τις προδιαγραφές,***
- ***πρέπει να αποδεικνύεται η ορθότητα του προγράμματος (δηλαδή η συμμόρφωση με τις προδιαγραφές).***

Αν αυτά που λέγαμε στις §0.3 και 0.4 σου φάνηκαν αφηρημένα και «ακαταλαβίστικα» ήρθε η ώρα να τα δεις σε εφαρμογή σε συγκεκριμένα παραδείγματα, να δεις πώς γίνεται η **επαλήθευση** (verification) ή **απόδειξη ορθότητας** (correctness proof) προγράμματος.

Ακόμη, στην §2.2 γράφαμε:

- ***Με ένα παράδειγμα ή με στιγμιότυπα εκτέλεσης μπορείς να ανακαλύψεις λάθη στο πρόγραμμά σου αλλά δεν μπορείς να αποδείξεις ότι το πρόγραμμα είναι σωστό.***

Πολλοί συγχέουν τις **δοκιμές προγράμματος** (program testing) με την απόδειξη ορθότητας· καμία σχέση! Μπορείς να κάνεις δοκιμές στο πρόγραμμά σου για να βρεις και να διορθώσεις λάθη που ενδεχομένως έχει. Το να περάσει το πρόγραμμα επιτυχώς όλες τις δοκιμές δεν σημαίνει ότι δεν έχει άλλα λάθη. Θα πεις «Και αν το δοκιμάσω επιτυχώς για όλες τις τιμές που μπορεί να πάρουν οι μεταβλητές του;» Ε, τότε εντάξει, αλλά τι πρόγραμμα είναι αυτό;

3.1 Ξεκινώντας με τη Δήλωση

Οι συνθήκες που ισχύουν μετά από τις δηλώσεις είναι αυτές που συνάγονται από τους Πιν. 2-1 και 2-2. Ας πούμε ότι κάνουμε τις δηλώσεις:

```
int          number;
unsigned int uNumber;
double       rNumber;
```

Όπως είπαμε «*έχουμε τη σιγουριά ότι η number έχει πάντοτε ακέραιη τιμή ανάμεσα στη μεγαλύτερη και τη μικρότερη που μπορεί να παρασταθεί στον τύπο* `int`» που μπορεί να γραφεί ώς εξής:

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

Αυτή είναι η **αναλλοίωτη** (invariant) του τύπου `int`.

Παρομοίως, για τον *uNumber* θα έχουμε:

$$(uNumber \in \mathbb{N}) \;\&\&\; (0 \le uNumber \le UINT\_MAX)$$[1]

και για τον *rNumber*:

$$(rNumber \in \mathbb{Q}) \;\&\&\; (-DBL\_MAX \le rNumber \le DBL\_MAX)$$

Οι συνθήκες αυτές ισχύουν «όσο ζουν» οι μεταβλητές.

Αν δώσουμε και αρχική τιμή; Π.χ.:

```
int number( 37 );
```

τότε θα έχουμε:[2]

$$(number == 37) \;\&\&\; (number \in \mathbb{Z}) \;\&\&\; (INT\_MIN \le number \le INT\_MAX)$$

Εδώ έχουμε πλεονασμό: αφού *number* `==` 37 προφανώς θα έχουμε και $number \in \mathbb{Z}$ και $INT\_MIN \le number \le INT\_MAX$. Δεν πειράζει· όπως είπαμε, αυτές που έχουν σχέση με τη δήλωση ισχύουν «όσο ζουν» οι μεταβλητές ενώ η "*number* `==` 37" μετά από λίγο μπορεί να μην ισχύει.

Πρόσεξε όμως και το εξής: αν δώσεις

```
int number( 3.0/2 );
```

μπορεί να πάρεις από τον μεταγλωττιστή μια *ειδοποίηση* (warning) της μορφής «`converting to 'int' from 'double'`» αλλά η *number* θα πάρει αρχική τιμή "`1`". Μπορούμε λοιπόν να πούμε ότι μετά την

```
int number( Π );
```

θα έχουμε:

---

[1] *UINT_MAX;;;* Έλυσες την άσκ. 2-12;

[2] Τι είναι πάλι αυτό το "`==`"; Επειδή το "`=`" στη C++ έχει συγκεκριμένο νόημα, αυτό που καθορίζεται στην εκχώρηση, χρησιμοποιούμε το "`==`" για τη σύγκριση «είναι ίσο με». Αυτό θα το ξανασυζητήσουμε.

(*number* == static_cast<int>(*Π*)) &&
(*number* ∈ ℤ) && (*INT_MIN* ≤ *number* ≤ *INT_MAX*)

Γενικώς, μπορούμε να πούμε ότι μετά την

```
T x( Π );
```

και αν ορίζεται η static_cast<*T*>(*Π*), θα έχουμε:

(*x* == static_cast<*T*>(*Π*)) && I_T(*x*)

όπου I_T είναι η *αναλλοίωτη* του τύπου *T*, δηλαδή η συνθήκη που ισχύει για όλες τις τιμές τύπου *T*.

3.2 Εντολές Εισόδου και Εξόδου

Εδώ τα πράγματα είναι απλά:

- Μια εντολή εξόδου δεν έχει επίδραση στις τιμές των μεταβλητών. Έτσι, αν ισχύει η *P* πριν από την

```
cout << number;
```

θα ισχύει και μετά την εκτέλεσή της.

- Μετά από μια εντολή εισόδου –αν η τιμή που εισάγεται μπορεί να παρασταθεί– δεν είναι δυνατόν να ξέρουμε κάτι περισσότερο από αυτό που ξέρουμε από τις δηλώσεις. Για παράδειγμα, με τις δηλώσεις της προηγούμενης παραγράφου, μετά την:

```
cin >> number;
```

θα έχουμε:

(*number* ∈ ℤ) && (*INT_MIN* ≤ *number* ≤ *INT_MAX*)

Οτιδήποτε άλλο ίσχυε για τη *number* (π.χ. κάτι σαν *number* == 37) παύει να ισχύει.

3.3 Οι Σταθερές στις Αποδείξεις

Πώς χειριστήκαμε τη σταθερά *g* στο παράδειγμα του προηγούμενου κεφαλαίου. *Είχαμε τη συνθήκη g* == 9.81 *αναλλοίωτη σε ολόκληρο το πρόγραμμα*.

Αυτό ακριβώς θα κάνουμε και με όλες τις σταθερές: αν έχουμε δηλώσει:

```
const T cn( c );
```

σε ολόκληρο το πρόγραμμά[3] μας ισχύει η *cn* == *c* και η αναλλοίωτη I_T(*cn*) του τύπου *T*:

(*cn* == *c*) && I_T(*cn*)

3.4 Το Αξίωμα της Εκχώρησης

Να δούμε τώρα πώς μπορούμε να διατυπώσουμε με ακρίβεια το νόημα της εντολής εκχώρησης. Ας πούμε ότι έχουμε δηλώσει

```
double x;
int y;
```

και έχουμε στο πρόγραμμά μας τις εντολές:

```
x = 7;      // x == 7.0
y = x + 4;  // y == 11
```

Σε σχόλια έχουμε βάλει το τι ξέρουμε ότι ισχύει σε εκείνο το σημείο της εκτέλεσης σύμφωνα με αυτά που είπαμε: «υπολογίζεται η τιμή της παράστασης, η τιμή μετατρέπεται στον τύπο της μεταβλητής, η τιμή φυλάγεται ως τιμή της μεταβλητής».

[3] Για την ακρίβεια: σε ολόκληρη την *εμβέλεια* της δήλωσης, όπως θα μάθουμε αργότερα.

Πάντως, στα προγράμματα που θα δούμε στη συνέχεια, το συνηθισμένο θα είναι να μην ξέρεις την τιμή της παράστασης· θα ξέρεις όμως μερικές ιδιότητές της, π.χ.: ας υποθέσουμε ότι σε κάποιο πρόγραμμα έχουμε δηλώσει

```
double x, a, b;
```

και έχουμε την εντολή:

```
x = pow( a + b, 2 ) + 1;
```

Πριν από την εκτέλεση της εντολής έχουμε a == a_0, b == b_0, αλλά τα a_0, b_0 μας είναι άγνωστα όταν γράφουμε το πρόγραμμα· ξέρουμε όμως ότι:

$$|a_0 + b_0| \le ½$$

Τι ξέρουμε για την τιμή της x μετά την εκτέλεση της εντολής; Η τιμή Π του δεξιού μέρους όταν εκτελείται η εκχώρηση είναι:

$$(a_0 + b_0)^2 + 1 = (|a_0 + b_0|)^2 + 1$$

και αφού ξέρουμε ότι: $|a_0 + b_0| \le ½$, θα έχουμε:

$$\Pi = (a_0 + b_0)^2 + 1 \le {}^5/_4$$

Μπορούμε λοιπόν να πούμε ότι μετά την εκτέλεση της εντολής εκχώρησης θα έχουμε:

$$x = (a_0 + b_0)^2 + 1 \le {}^5/_4$$

Να λοιπόν τι μπορούμε να πούμε γενικά:

- Αν
  - η `v` είναι τύπου T και
  - πριν από την εκτέλεση της `v` = Π ισχύει η $P($ `static_cast<`T`>(`Π`)` $)$ και
- Αν η εκτέλεση της `v` = Π τερματισθεί κανονικά

τότε

- Μετά την εκτέλεση της `v` = Π ισχύει η $P(v)$, δηλαδή η συνθήκη που προκύπτει από στην $P($ `static_cast<`T`>(`Π`)` $)$ αν αντικαταστήσουμε με τη v τη `static _cast<`T`>(` Π `)`.

Αυτό είναι το **αξίωμα της εκχώρησης**.

Παρατηρήσεις: ▶

1. Η $P($ `static_cast<`T`>(`Π`)` $)$ μπορεί να περιλαμβάνει και συνθήκες που δεν περιλαμβάνουν τη v. Αυτές δεν επηρεάζονται από την εντολή εκχώρησης, αλλά παραμένουν αναλλοίωτες από αυτήν. Στο παράδειγμά μας, συμφώνως με όσα είπαμε, θα έχουμε πριν από την εντολή:

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

Όλα αυτά δεν επηρεάζονται από την εκτέλεση της

```
x = pow( a + b, 2 ) + 1;
```

και ισχύουν και μετά από αυτήν.

2. Ό,τι ίσχυε για τη v πριν από την εκτέλεση της "`v` = Π" δεν ισχύει μετά από αυτήν. Γυρνώντας στο παράδειγμά μας, ας πούμε ότι πριν από την εκτέλεση της `x = pow(a + b, 2) + 1` έχουμε x == 0. Μετά την εκτέλεσή της, το μόνο από τα «παλιά» που μένει να ισχύει είναι η $-DBL\_MAX \le x \le DBL\_MAX$ που απορρέει από τη δήλωση `double x`. Η x == 0 παύει να ισχύει και αντικαθίσταται από την $x = (a_0 + b_0)^2 + 1 \le {}^5/_4$. ◀

Μπορούμε να δούμε το αξίωμα της εκχώρησης και με έναν άλλο τρόπο:

- Αν η εκτέλεση της `v` = Π τερματισθεί κανονικά,
- Για να ισχύει μετά την εκτέλεση της `v` = Π η $P(v)$

θα πρέπει

- πριν από την `v` = Π να ισχύει η $P($ `static_cast<`T`>(`Π`)` $)$, δηλαδή η συνθήκη που προκύπτει αν στην $P(v)$ βάλουμε όπου v τη `static_cast<`T`>(` Π `)`.

Όπως θα δεις στη συνέχεια, αυτόν τον τρόπο θα χρησιμοποιούμε συνήθως.

3.5 Συνθήκες "true" και "false"

Πολύ συχνά θέλουμε να γράψουμε τη συνθήκη: «για οποιαδήποτε τιμή των μεταβλητών μας». Αυτή γράφεται ως εξής: `true`. Να ένα παράδειγμα τέτοιας συνθήκης:

$(x \geq 0)$ `||` $(x < 0)$

Όπως υπάρχει συνθήκη `true`, υπάρχει και συνθήκη `false` και σημαίνει: «δεν ισχύει όποιες και αν είναι οι τιμές των μεταβλητών μας.» Π.χ.

$(x \geq 0)$ `&&` $(x < 0)$

3.6 Το Πρόγραμμα Μαζί με την Απόδειξη

Και τώρα να δούμε, με παραδείγματα, πώς χρησιμοποιούμε το αξίωμα της εκχώρησης για να αποδείξουμε την ορθότητα ενός προγράμματος. Στην πραγματικότητα θα δούμε τρεις φορές, από τρεις διαφορετικές σκοπιές, το παράδειγμα της αντιμετάθεσης. Όπως είπαμε όμως, η ορθότητα του προγράμματος αναφέρεται σε συγκεκριμένες **προδιαγραφές** (program specifications). Το πρώτο πράγμα που πρέπει να κάνουμε είναι να διατυπώσουμε αυτές τις προδιαγραφές για το πρόβλημά μας. Αυτό είναι απλό:

Προϋπόθεση: $(a == a_0)$ `&&` $(b == b_0)$

Απαίτηση: $(a == b_0)$ `&&` $(b == a_0)$

που σημαίνουν: Αν πριν από την πρώτη εντολή ισχύει η συνθήκη $(a == a_0)$ `&&` $(b == b_0)$ τότε μετά την εκτέλεση όλων των εντολών θα ισχύει: $(a == b_0)$ `&&` $(b == a_0)$.

Πώς γίνεται *πρακτικά* να πάρουμε την απόδειξη ορθότητας ενός προγράμματος; Θα δούμε πώς γίνεται αυτό από τρεις διαφορετικές «απόψεις»:

- Γράφουμε το πρόγραμμα και –*παράλληλα*– καταγράφουμε τις συνθήκες που προκύπτουν από την εκτέλεση της κάθε εντολής.
- Μας δίνεται το πρόγραμμα (ή το γράφουμε εμείς) και μετά αποδεικνύουμε ότι είναι σωστό.
- Γράφουμε το πρόγραμμα προσπαθώντας να βάζουμε τις εντολές που θα μας δώσουν αυτά που ζητούν οι προδιαγραφές.

Θα δοκιμάσουμε αυτές τις τρεις «απόψεις» στο πρόγραμμα της *αντιμετάθεσης*. Και στις τρεις περιπτώσεις, θα παρεμβάλουμε, μέσα σε σχόλια, τις συνθήκες που μας ενδιαφέρουν.

3.6.1 Απόδειξη – Πρόγραμμα Παραλλήλως

Ας δούμε αρχικά την απόδειξη που γίνεται παραλλήλως με το γράψιμο. Στην περίπτωση αυτή θα πρέπει να βάζουμε μετά από κάθε εντολή τη συνθήκη που προκύπτει από την εκτέλεσή της. Θα τη βρίσκουμε με την εξής συνταγή που βγάζουμε από το αξίωμα της εκχώρησης:

- ***Αν έχεις P(`static_cast<T>(Π)`) { `v = Π` } P(v) θα πάρεις την P(v) ως εξής:***
  1. ***Θα αντιγράψεις την P(`static_cast<T>(Π)`).***
  2. ***Θα σβήσεις όποιαδήποτε συνθήκη περιλαμβάνει τη v. Η τιμή της v άλλαξε και ό,τι ίσχυε γι' αυτήν παύει να ισχύει.***
  3. ***Οτιδήποτε υπάρχει για την Π θα γραφεί άλλη μια φορά –με `&&`– με αντικατάσταση της Π από την v.***

Ξεκινούμε λοιπόν με:

```
// (a == a0) && (b == b0)
```

Το πρώτο που κάναμε ήταν να φυλάξουμε κάπου (στην *s*) την αρχική τιμή της *a*. Αυτό γίνεται με την εντολή εκχώρησης `s = a`. Τι θα ισχύει μετά την εκτέλεση αυτής της εντολής;

Ας βρούμε τη συνθήκη που θα ισχύει μετά την εντολή. Στην περίπτωσή μας μεταβλητή είναι η *s* και παράσταση είναι η *a*. Άρα:

1. Παίρνουμε τη συνθήκη που ισχύει πριν από αυτήν: (a == a_0) && (b == b_0).
2. Αφού δεν υπάρχει τίποτε για την *s*, δεν κάνουμε διαγραφές.
3. Για την *a* υπάρχει η a == a_0· την αντιγράφουμε αντικαθιστώντας την *a* με *s*:

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
```

Ύστερα από αυτό, μπορούμε να βάλουμε στην *a* την τιμή της *b* (`a = b`). Πώς παίρνουμε τη συνθήκη που θα ισχύει μετά την εντολή. Εδώ, μεταβλητή είναι η *a* και παράσταση είναι η *b*. Σύμφωνα με τη συνταγή μας:

1. Παίρνουμε τη συνθήκη που ισχύει πριν από αυτήν: (a == a_0) && (b == b_0) && (s == a_0).
2. Διαγράφουμε από αυτήν ό,τι ισχύει για την *a*: (b == b_0) && (s == a_0).
3. Για τη *b* υπάρχει η b == b_0· την αντιγράφουμε αντικαθιστώντας τη *b* με *a*:

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
   a = b;
// (a == b0) && (b == b0) && (s == a0)
```

Τέλος, δίνουμε στη *b* την τιμή της *a*, που έχουμε «φυλάξει» στην *s*. Ό,τι ίσχυε για τη *b* (b == b_0) παύει να ισχύει· από εδώ και πέρα για την *b* ισχύει ό,τι ίσχυε μέχρι τώρα την *s* (b == a_0):

```
// (a == a0) && (b == b0)
   s = a;
// (a == a0) && (b == b0) && (s == a0)
   a = b;
// (a == b0) && (b == b0) && (s == a0)
   b = s
// (a == b0) && (b == a0) && (s == a0)
```

Αποδείξαμε ότι το πρόγραμμα είναι σωστό; Η συνθήκη που πήραμε τελικώς δεν είναι η απαίτηση. Αλλά... για να δούμε: Αποδείξαμε ότι αν το πρόγραμμά μας εκτελεσθεί με προϋπόθεση (a == a_0) && (b == b_0), θα καταλήξουμε στη συνθήκη: (a == b_0) && (b == a_0) && (s == a_0). Στο Παρ. Α, §Α.4, βλέπουμε ότι (P && Q) ⇒ P· στην περίπτωσή μας, αν πάρουμε ως P την (a == b_0) && (b == a_0) και ως Q την (s == a_0) έχουμε:

$$((a == b_0)\ \&\&\ (b == a_0))\ \&\&\ (s == a_0) \Rightarrow ((a == b_0)\ \&\&\ (b == a_0))$$

Αλλά ο συμπερασματικός κανόνας (*E*1), που είδαμε στην §0.4.1, λέει ότι αποδείξαμε την ορθότητα του προγράμματός μας.

3.6.2 Απόδειξη εκ των Υστέρων (από το Τέλος προς την Αρχή)

Τώρα, ας έρθουμε στη δεύτερη περίπτωση: Γράψαμε (ή μας δίνεται) το πρόγραμμα και πρέπει να αποδείξουμε ότι είναι σωστό σε σχέση με τις προδιαγραφές του. Μπορούμε να κάνουμε την απόδειξη είτε *από την αρχή προς το τέλος* είτε *από το τέλος προς την αρχή*.

Για το παράδειγμα της αντιμετάθεσης, αν κάνουμε την απόδειξη από την αρχή προς το τέλος, θα πάρουμε την απόδειξη που είδαμε παραπάνω.

Για να κάνουμε μια απόδειξη *από το τέλος προς την αρχή* θα πρέπει να μπορούμε να απαντούμε στο εξής ερώτημα: Αν ξέρω τη συνθήκη Q που ισχύει μετά την εκτέλεση της εντολής E, πώς μπορώ να βρω τη συνθήκη που πρέπει να ισχύει πριν από αυτήν; Χρησιμοποιώντας τη δεύτερη διατύπωση του αξιώματος της εκχώρησης, που είδαμε στο τέλος της §3.4, βγάζουμε την εξής συνταγή:

- ***Αν έχεις P(`static_cast<T>(Π)`) { `v = Π` } P(v) θα πάρεις την P(`static_cast<T>(Π)`) ως εξής:***

1. *Θα αντιγράψεις την P(v).*
2. *Οπου υπάρχει η v θα την αντικαταστήσεις με τη* `static_cast<T>(Π)`.

🡒🡒🡒

Στο παράδειγμά μας θα πρέπει να αποδείξουμε:

```
// (a == a0) && (b == b0)
   s = a;  a = b;  b = s;
// (a == b0) && (b == a0)
```

Ξεκινούμε από την:

```
   b = s;
// (a == b0) && (b == a0)
```

Πώς παίρνουμε τη συνθήκη που θα ισχύει πριν από την εντολή; Εδώ, μεταβλητή είναι η *b* και παράσταση είναι η *s*. Σύμφωνα με τη συνταγή μας:

1. Παίρνουμε τη συνθήκη που ισχύει μετά την εντολή: (a == b_0) && (b == a_0).
2. Αντικαθιστούμε σε αυτήν όπου *b* την *s*: (a == b_0) && (s == a_0).

Φθάνουμε λοιπόν στο εξής:

```
   s = a;  a = b;
// (a == b0) && (s == a0)
   b = s;
// (a == b0) && (b == a0)
```

Με τον ίδιο τρόπο βρίσκουμε τη συνθήκη που πρέπει να ισχύει πριν από την `a = b`. Μεταβλητή είναι η *a* και παράσταση είναι η *b*. Άρα:

1. Παίρνουμε τη συνθήκη που ισχύει μετά την εντολή: (a == b_0) && (s == a_0).
2. Αντικαθιστούμε σε αυτήν όπου *a* τη *b*: (b == b_0) && (s == a_0).

Έχουμε λοιπόν:

```
   s = a;
// (b == b0) && (s == a0)
   a = B;
// (a == b0) && (s == a0)
   B = s;
// (a == b0) && (b == a0)
```

Τέλος, ας βρούμε τη συνθήκη που θα πρέπει να ισχύει πριν από την `s = a` ώστε μετά από αυτήν να έχουμε (b == b_0) && (s == a_0). Μεταβλητή μας είναι η *s* και παράσταση η *a*. Άρα:

1. παίρνουμε την (b == b_0) && (s == a_0) και
2. αντικαθιστούμε το *s* με το *a* και παίρνουμε την (b == b_0) && (a == a_0).

Φθάνουμε λοιπόν στο εξής:

```
// (b == b0) && (a == a0)
   s = a;
// (b == b0) && (s == a0)
   a = b;
// (a == b0) && (s == a0)
   b = s;
// (a == b0) && (b == a0)
```

Για να είναι το πρόγραμμά μας σωστό θα πρέπει: η συνθήκη (b == b_0) && (a == a_0) –που θέλουμε να ισχύει πριν από την πρώτη εντολή– να συνάγεται από την προϋπόθεση. Αλλά αυτό συμβαίνει: Επειδή η πράξη && είναι αντιμεταθετική, από την προϋπόθεση (a == a_0) && (b == b_0) παίρνουμε τη (b == b_0) && (a == a_0). Άρα το πρόγραμμά μας είναι σωστό.

🡐🡐🡐

3.6.3 Το Πρόγραμμα από τις Προδιαγραφές

Τέλος, ας έρθουμε στην τρίτη «άποψη»: Να γράψουμε το πρόγραμμα με οδηγό τις προδιαγραφές; Θα ρωτήσεις: Δηλαδή μπορούμε να το γράψουμε και αλλιώς; Όχι βέβαια·

αυτό που εννοούμε είναι να δουλέψουμε πιο συστηματικά και από τις απαιτήσεις να βρίσκουμε τις εντολές.

Θα δουλέψουμε και πάλι από το τέλος προς την αρχή.

✎✎✎

Στο παράδειγμά μας ξεκινούμε με την ερώτηση: με ποια εντολή μπορούμε να καταλήξουμε στη συνθήκη:

```
// (a == b0) && (b == a0)
```

Σύμφωνα με το αξίωμα της εκχώρησης μπορούμε να πάρουμε την b `==` a_0, αν εκτελεσθεί η εντολή `b = s` και πριν από αυτήν ισχύει η s `==` a_0. Παρόμοια, μπορούμε να φτάσουμε στην a = b_0, αν εκτελεσθεί η εντολή `a = t` και πριν από αυτήν ισχύει η t `==` b_0:

```
// (t == b0) && (s == a0)
   a = t;  b = s;
// (a == b0) && (b == a0)
```

Πώς μπορούμε να φτάσουμε στην (t `==` b_0) `&&` (s `==` a_0); Σύμφωνα με το αξίωμα της εκχώρησης και πάλι, μπορούμε να φτάσουμε στη συνθήκη αυτή αν εκτελεσθούν οι εντολές `s = a; t = b` και πριν από αυτές ισχύει η (a `==` a_0) `&&` (b `==` b_0), που είναι η προϋπόθεσή μας. Άρα, το παρακάτω πρόγραμμα είναι σωστό:

```
// (a == a0) && (b == b0)
   s = a;  t = b;
// (t == b0) && (s == a0)
   a = t;  b = s;
// (a == b0) && (b == a0)
```

Εδώ χρησιμοποιήσαμε δύο βοηθητικές μεταβλητές, αλλά αυτό δεν είναι κάτι τραγικό.

✎✎✎

3.7 Διά Ταύτα ...

Πριν κάνουμε μια σύγκριση των τριών απόψεων να προλάβουμε κάποιους που μπορεί να κάνουν ήδη απαισιόδοξες σκέψεις του εξής τύπου: «Και αν έχω ένα πρόγραμμα των εκατό γραμμών τι γίνεται; Θα έχω συνθήκες που πιάνουν σελίδες! Για να μη πούμε τι θα γίνει όταν, όπως γίνεται στην πραγματικότητα, έχουμε χιλιάδες γραμμές προγράμματος!» Η απάντηση είναι: τα προγράμματα γράφονται με τη διαδικασία της **βήμα-προς-βήμα ανάλυσης** (§0.5). Έτσι τα κομμάτια προγράμματος που πρέπει να γραφούν και να αποδειχτούν έχουν μέγεθος και «δυσκολία» που μπορούμε να διαχειριστούμε[4]. Αυτά ισχύουν και για τον αντικειμενοστραφή προγραμματισμό· εκεί όμως έχουμε διαφορετικό τρόπο καταμερισμού της «δυσκολίας».

Ποιος από τους τρεις τρόπους εργασίας είναι η πιο συνηθισμένος; Ο δεύτερος και μάλιστα με απόδειξη από το τέλος προς την αρχή! Δηλαδή, ο προγραμματιστής –με τη βήμα-προς-βήμα ανάλυση ή με άλλον τρόπο– έχει φτάσει σε ένα πρόβλημα που από τις προδιαγραφές του «φαίνεται» η λύση. Γράφει λοιπόν τη λύση και μετά αποδεικνύει ότι είναι σωστή.

Ο πρώτος, όπως και ο δεύτερος, αλλά με απόδειξη από την αρχή προς το τέλος, είδες ότι έχουν πιο πολύπλοκη συνταγή για την εύρεση των συνθηκών που μας χρειάζονται. Έχει όμως και το εξής πρόβλημα: Οι συνθήκες «φουσκώνουν» πολύ γρήγορα –με κομμάτια που είναι άχρηστα στη συνέχεια– και γίνονται δύσχρηστες. Στο παράδειγμά μας είδες ότι καταλήξαμε με την (άχρηστη) s `==` a_0 μετά το τελευταίο βήμα.

Ο τρίτος έχει ενδιαφέρον όταν προσπαθούμε να λύσουμε το εξής πρόβλημα: Να γραφεί ένα πρόγραμμα που θα τροφοδοτείται με τις προδιαγραφές ενός προγράμματος και θα το γράφει, ας πούμε σε C++ (αυτόματη σύνθεση προγράμματος). Φυσικά, αν μπορέσουμε να

[4]Πάντως, συνθήκες που πιάνουν αρκετές γραμμές είναι συνηθισμένες.

γράψουμε αλγόριθμο που να κάνει κάτι τέτοιο, θα μπορούμε, πολύ πιο εύκολα να διατυπώσουμε κανόνες που να τους μάθει ένας άνθρωπος.

Εμείς θα χρησιμοποιούμε κατά κύριο λόγο τον δεύτερο τρόπο, με απόδειξη από το τέλος προς την αρχή και σπανιότερα τους άλλους.

Κάποιοι θα αναρωτηθούν «όταν τελειώσει η απόδειξη, θα σβήσουμε τα σχόλια με τις συνθήκες;» Καλύτερα να τις κρατάμε, τουλάχιστον ορισμένες από αυτές. Οι **συνθήκες επαλήθευσης** (verification conditions) είναι απαραίτητες σε περίπτωση που θα θελήσουμε να αλλάξουμε κάτι αργότερα.

3.8 Τα Προβλήματα των Τύπων Κινητής Υποδιαστολής

Πριν προσπαθήσουμε να κάνουμε μια απόδειξη προγράμματος με πραγματικούς αριθμούς (αυτού για την ελεύθερη πτώση της §2.7) ας δούμε μερικά (ακόμη) προβλήματα των τύπων κινητής υποδιαστολής.

Όπως λέγαμε στην §2.5 «*Η παράσταση των τιμών τύπου* `double` *(και* `float` *και* `long double`*) γίνεται προσεγγιστικώς.*» και «*οι πράξεις ... στους πραγματικούς τύπους (όπως ο* `double`*) δεν είναι* [ακριβείς]». Ήδη, στην §1.7.1 είδαμε ότι η:

```
cout.precision(20);
cout << 12.34567890123456789 << endl;
```

μας δίνει:

```
12.345678901234567884
```

Μια ματιά στον Πίν. 2-2 μας λέει ότι η ακρίβεια που μπορούμε να ζητήσουμε από τον `double` δεν υπερβαίνει τα 16 ψηφία. Φυσικά και στην περίπτωση αυτή έχουμε:

```
12.34567890123457
```

Ας προσπαθήσουμε να υπολογίσουμε δυνάμεις με τη χρήση της pow:

```
v = pow(0.3,4.0);   cout << " 0.3^4 = " << v << endl;
v = pow(0.7,2.0);   cout << " 0.7^2 = " << v << endl;
```

Αποτελέσματα:

```
0.3^4 = 0.0081
0.7^2 = 0.4899999999999999
```

Το δεύτερο είναι εντυπωσιακό.

Αν προσπαθήσουμε να κάνουμε το ίδιο πράγμα με τιμές τύπου `float` (εδώ βέβαια η ακρίβεια δεν μπορεί να είναι μεγαλύτερη από 8 ψηφία):

```
cout.precision(8);
v = pow(0.3f,4.0);  cout << " 0.3^4 = " << v << endl;
v = pow(0.7f,2.0);  cout << " 0.7^2 = " << v << endl;
```

θα πάρουμε:

```
0.3^4 = 0.0081000013
0.7^2 = 0.48999998
```

Τι συμπέρασμα βγαίνει από τα παραπάνω; Στις προδιαγραφές, όπου έχουμε τιμές πραγματικού τύπου (π.χ. `double`), δεν μπορούμε να βάζουμε ισότητα.

Στο παράδειγμα της ελεύθερης πτώσης γράψαμε:

Απαίτηση: $(tP == \sqrt{2h/g}\,)$ && $(vP == -\sqrt{2gh}\,)$

αλλά πιο πολύ νόημα θα είχε αν γράφαμε:

Απαίτηση: $(tP \approx \sqrt{2h/g}\,)$ && $(vP \approx -\sqrt{2gh}\,)$

Αν θέλουμε να γράψουμε κάτι πιο ποσοτικό θα μπορούσαμε να βάλουμε φράγματα στο απόλυτο σφάλμα:

Απαίτηση: $(|tP - \sqrt{2h/g}\,| < \varepsilon_{TA})$ && $(|vP + \sqrt{2gh}\,| < \varepsilon_{VA})$

όπου ε_{TA} και ε_{VA} θετικοί αριθμοί, *φράγματα στο απόλυτο σφάλμα*. Τι σημαίνει; Η τιμή της *tP* πρέπει να διαφέρει από την ακριβή τιμή της $\sqrt{2h/g}$ λιγότερο από ε_{TA} και η τιμή της *vP* να διαφέρει από την ακριβή τιμή της, $-\sqrt{2gh}$, λιγότερο από ε_{VA}.

Παρομοίως, μπορούμε να βάλουμε φράγμα στο σχετικό σφάλμα:

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

Τα ε_{TA}, ε_{VA}, ε_{TR} και ε_{VR} εξαρτώνται όχι μόνο από τον τύπο κινητής υποδιαστολής που χρησιμοποιούμε αλλά και (κυρίως) από το πρόβλημα που λύνουμε. Στο παράδειγμά μας δεν έχει νόημα ακρίβεια μεγαλύτερη από τρία ψηφία. Αυτό μεταφράζεται σε $\varepsilon_{TR} = \varepsilon_{VR} = 5\times10^{-4}$. Φυσικά, ακόμη και ο `float` μας εγγυάται πολύ καλύτερη ακρίβεια.

Και τι θα κάνουμε εδώ; Θα συνεχίσουμε να δίνουμε παραδείγματα στον τύπο `double`, αλλά στις προδιαγραφές θα βάζουμε είτε φράγματα στο σφάλμα, όπως παραπάνω, είτε το "≈" αντί για "`==`". Εσύ θα πρέπει να θυμάσαι ότι *οι πραγματικοί τύποι θέλουν ιδιαίτερη προσοχή*.

3.9 Μια Απόδειξη με Πραγματικούς

Και τώρα να δούμε: Είναι σωστό το πρόγραμμά για την ελεύθερη πτώση; Θα πάρουμε ως προδιαγραφές:

Προϋπόθεση: $(g == 9.81)$ **&&** $(h \ge 0)$

Απαίτηση: $(tP \approx \sqrt{2h/g})$ **&&** $(vP \approx -\sqrt{2gh})$

Θα πρέπει δηλαδή να αποδείξουμε:

```
// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   tP = sqrt(2*h/g);
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Θα κάνουμε την απόδειξη από το τέλος προς την αρχή και ξεκινάμε από την:

```
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Ποια συνθήκη θα πρέπει να ισχύει πριν από την εκτέλεση της εντολής; Σύμφωνα με τη συνταγή μας, καταλήγουμε στην:

$$(tP \approx \sqrt{2h/g}) \;\&\&\; (-g\cdot tP \approx -\sqrt{2hg})$$

Φτάσαμε λοιπόν στο:

```
   tP = sqrt(2*h/g);
// (tP ≈ √(2h/g)) && (gTp ≈ √(2hg))
   vP = -g*tP;
// (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
```

Τι θα πρέπει να έχουμε πριν από την `tP = sqrt(2*h/g)`; Και πάλι σύμφωνα με τη συνταγή μας βρίσκουμε:

$$(sqrt(2h/g) \approx \sqrt{2h/g}) \;\&\&\; (g\cdot sqrt(2h/g) \approx \sqrt{2hg})$$

Εδώ όμως πρέπει να προσέξουμε και κάτι άλλο:

- έχουμε διαίρεση δια *g* και θα πρέπει να έχουμε $g \ne 0$,
- έχουμε τετραγωνικές ρίζες και για να υπολογίζονται θα πρέπει να έχουμε: $2h/g \ge 0$ και $2hg \ge 0$ (η μια από τις δύο φτάνει).

  Θα πρέπει λοιπόν να έχουμε:

$$(g \ne 0) \;\&\&\; (hg \ge 0) \;\&\&\; (sqrt(2h/g) \approx \sqrt{2h/g}) \;\&\&\; (g\cdot sqrt(2h/g) \approx \sqrt{2hg})$$

  Από την προϋπόθεση, $(g == 9.81)$ **&&** $(h \ge 0)$

- έπεται η $hg \ge 0$ και
- από τη $g == 9.81$ έπεται η $g \ne 0$

Αν πούμε ότι η *sqrt* υπολογίζει μια, όσο γίνεται καλύτερη, *προσέγγιση* της τετραγωνικής ρίζας, ισχύει και η $(sqrt(2h/g) \approx \sqrt{2h/g})$ && $(g \cdot sqrt(2h/g) \approx \sqrt{2hg})$.

Πρόσεξε ακόμη ότι αν δεν έχουμε τη g == 9.81 οι υπολογισμοί μας χάνουν το φυσικό τους νόημα, αφού τα αποτελέσματα που έχουμε από τη Φυσική θέλουν αυτήν την τιμή της g.

Προσοχή! Το αν θα ισχύει η $h \geq 0$ εξαρτάται από την καλή πρόθεση του χρήστη. Αργότερα θα μάθουμε πώς μπορούμε να κάνουμε *έλεγχο* της προϋπόθεσης και να ενεργήσουμε ανάλογα.

Αλλά τώρα μας έρχονται και άλλες ιδέες: ας πούμε ότι το h δεν είναι αρνητικό· μήπως υπάρχει περίπτωση να είναι πολύ μεγάλο και να έχουμε υπερχείλιση; Ας το σκεφτούμε. Για το h θα έχουμε: $0 \leq h \leq DBL\_MAX$. Όταν πάει να εκτελέσει την εντολή:

```
tP = sqrt(2*h/g);
```

για το πολλαπλασιασμό θα έχουμε:

$$0 \leq 2h \leq 2DBL\_MAX$$

Να μια περίπτωση υπερχείλισης[5] και μάλιστα χωρίς λόγο, διότι αν κάνουμε πρώτα τη διαίρεση δια g όλα πάνε καλά:

$$(2/g)h \leq (2/g)DBL\_MAX < DBL\_MAX, \text{ αφού } 2/g < 1$$

Θα πρέπει λοιπόν να υποχρεώσουμε, π.χ. με χρήση παρενθέσεων, τον υπολογιστή να κάνει τις πράξεις όπως εμείς θέλουμε:

```
tP = sqrt((2/g)*h);
```

Ύστερα από τα παραπάνω, η τιμή της *tP* θα είναι:

$$0 \leq tP == sqrt((2/g)h) \leq \sqrt{\frac{2}{g}}\sqrt{DBL\_MAX}$$

Ερχόμαστε τώρα στην:

```
vP = -g*tP;
```

Από την σχέση για το *tP*, πολλαπλασιάζοντας επί g, παίρνουμε:

$$0 \leq g \cdot sqrt((2/g)h) \leq g\sqrt{\frac{2}{g}}\sqrt{DBL\_MAX}$$

Αλλά, η $g\sqrt{2/g}\sqrt{DBL\_MAX} == \sqrt{2g}\sqrt{DBL\_MAX}$ είναι μικρότερη από *DBL_MAX* όταν έχουμε: $2g < DBL\_MAX$. Αυτό ισχύει για οποιονδήποτε τύπο `double`, αφού οι συνηθισμένες τιμές της *DBL_MAX* είναι της τάξης του 10^{308}. Άρα και η $-DBL\_MAX \leq g \cdot sqrt(2h/g) \leq DBL\_MAX$ ισχύει.

Βλέπουμε κοιπόν ότι η μελέτη της ορθότητας του προγράμματος μας οδηγεί σε διορθώσεις όπως αυτή της εντολής: `tP = sqrt(2*h/g)` που την αλλάξαμε σε: `tP = sqrt((2/g)/h)`. Και πρόσεξε και κάτι ακόμη: αν δεν γράφαμε τη `vP = -g*tP` για λόγους «οικονομίας», αλλά τη βάζαμε `vP = -sqrt(2*g*h)`, θα είχαμε και εδώ προβλήματα «υπερχείλισης χωρίς λόγο».

3.10 Τι Να Κάνουμε

Σε γενικές γραμμές έχουμε τρεις τρόπους για να αποδείξουμε την ορθότητα ενός προγράμματος. Συνήθως την αποδεικνύουμε πηγαίνοντας από το τέλος προς την αρχή. Αργότερα θα δούμε ότι και οι άλλοι δύο τρόποι είναι χρήσιμοι σε πολλές περιπτώσεις.

Τα εργαλεία που χρησιμοποιούμε στην απόδειξη είναι

- *αξιώματα* (όπως αυτό της εκχώρησης) και

[5] «Σιγά που θα έχουμε υπερχείλιση!» σκέφτεσαι «Για να γίνει κάτι τέτοιο θα πρέπει να έχουμε $h > DBL\_MAX/2$ και για τέτοιες τιμές δεν ισχύουν αυτοί οι τύποι.» Σωστό! Αλλά αυτά είναι δουλειά της Φυσικής· εδώ κοιτάμε μόνο τις προδιαγραφές μας.

- *συμπερασματικοί κανόνες* όπως αυτοί που είδαμε στο Κεφ. 0.

Ιδιαίτερη προσοχή θα πρέπει να δίνουμε σε «κακοτοπιές» όπως η υπερχείλιση, η κλήση συνάρτησης με όρισμα εκτός πεδίου ορισμού (π.χ. αρνητικό υπόριζο) και άλλα παρόμοια.

Θα πρέπει να αποδεικνύουμε την ορθότητα όλων των προγραμμάτων που γράφουμε; Κατ' αρχήν «Ναι!» αλλά αυτό δεν είναι και τόσο εύκολο. Αυτό που κάνουμε συνήθως είναι να επαληθεύουμε ορισμένα κρίσιμα κομμάτια του κάθε προγράμματος.

Ασκήσεις

Α Ομάδα

3-1 Απόδειξε ότι: Αν

```
   int k, sum;
```

τότε:

```
// sum == (k - 1)*k/2
   sum = sum + k;
   k = k + 1;
// sum == (k - 1)*k/2
```

3-2 Απόδειξε ότι: Αν

```
   double x, y, z;
```

τότε:

```
// z == y/2 - 1
   x = z + 1;   x = x + y/2;   z = z + 1;
// (x == y) && (z == y/2) }
```

Β Ομάδα

3-3 Αν, στο πρόβλημα της αντιμετάθεσης, ο τύπος των μεταβλητών είναι αριθμητικός, υπάρχει και άλλη λύση χωρίς βοηθητική μεταβλητή:

```
   a = a - b;  b = b + a;  a = b - a;
```

Απόδειξε ότι είναι σωστός. Αγνόησε την περίπτωση υπερχείλισης.

3-4 Απόδειξε ότι: Αν

```
   int d1, d2, q, r;
```

τότε:

```
// (r ≥ d2) && (r ≥ 0) && (d1 = q*d2 + r)
   q = q + 1;
   r = r - d2;
// (r ≥ 0) && (d1 == q*d2 + r)
```

3-5 Απόδειξε ότι: Αν

```
   int k, x, y;
```

τότε:

```
// true
   k = 0;  x = 1;  y = 1;
// (y = 2k+1) && (x == (k+1)²)
```

και

```
// (y == 2k+1) && (x == (k+1)²)
   k = k + 1;  y = y + 2;  x = x + y;
// (y == 2k+1) && (x == (k+1)²)
```

Γ Ομάδα

3-6 Τι σχέση υπάρχει μεταξύ (`double`) x και `static_cast<long>` (x); Διατύπωσε τις προδιαγραφές της συνάρτησης `static_cast<long>` (ή, τουλάχιστον, προσπάθησε).
Υπόδ.: Ξεχώρισε τις περιπτώσεις $x \geq 0$ και $x < 0$.

3-7 Δίνεται το πρόβλημα: Γράψε εντολές που να μεταθέτουν κυκλικώς τις τιμές τριών μεταβλητών a, b, c ($a \leftarrow b \leftarrow c \leftarrow a$). Διατύπωσε τις προδιαγραφές του και γράψε τις εντολές που το λύνουν μαζί με την απόδειξη ορθότητας.