HY255 Εργαστήριο Λογισμικού – L14

Checking error conditions
(assertions, runtime checked errors)

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

Συνθήκες λάθους – Error conditions

- Ας χωρίσουμε τις συνθήκες λάθους σε δύο κατηγορίες
- (1) Συνθήκες λάθους που δεν πρέπει να συμβούν, by design → Ας τις ονομάσουμε υποθέσεις (assertions)
- (2) Συνθήκες που μπορεί να συμβούν, ακόμη και αν είναι κάποια περίπτωση που το πρόγραμμά μας θα χειριστεί με απλό τρόπο, πχ να κάνει exit → Ας τις ονομάσουμε ελεγχόμενες συνθήκες (runtime checked errors)
- Ας εξετάσουμε την κάθε κατηγορία χωριστά

(1) Υποθέσεις (Assertions)

- Κάθε κομμάτι κώδικα έχει κάποιο input με την γενική έννοια, δηλαδή περιμένει η συνολική κατάσταση του προγράμματος (μνήμη) να είναι σε κάποια μορφή
  - Σε μια συνάρτηση χωρίς side-effects το input είναι μόνο οι παράμετροι της συνάρτησης
  - Σε ένα μια συνάρτηση με global μεταβλητές ή/και χρήση του heap η κατάσταση περιλαμβάνει και αυτές τις τιμές
- Όταν τελειώσει η εκτέλεση αυτού του κομματιού κώδικα, θα αφήσει την κατάσταση του προγράμματος σε κάποια νέα μορφή, πχ θα έχει αλλάξει τις τιμές μεταβλητών, θα έχει δεσμεύσει μνήμη κοκ
- Το σημαντικό σε αυτό το σημείο είναι η συνειδητοποίηση ότι για να γράψουμε σωστά αυτό το κομμάτι κώδικα θα πρέπει να γνωρίζουμε (εμείς που το σχεδιάζουμε) ποια είναι η input κατάσταση και ποια θα πρέπει να είναι η output κατάσταση
- Αν αυτό μας είναι ξεκάθαρο, τότε μπορούμε να σχεδιάσουμε τον κώδικά μας ώστε να μας μεταφέρει από την αρχική στην τελική κατάσταση και να είναι πάντα σωστή η εκτέλεσή του προγράμματός μας
- Π.χ. ας σκεφτούμε την συνάρτηση strcpy από την 2η άσκηση (εκ των υστέρων και αφού την έχουμε γράψει)

```
char *strcpy(char *s1, const char *s2) {
        char *s = s1;
        while ( (*(s++) = *(s2++)) != '\0') ;
        return s1;
}
```

- Τι θεωρήσαμε ότι είναι η input/αρχική κατάσταση όταν γράψαμε το κομμάτι αυτό κώδικα και τι θέλουμε να είναι η output/τελική κατάσταση όταν τελειώσει ο κώδικας αυτός;
- Πως μπορούμε να εξασφαλίσουμε ότι αυτό το κομμάτι κώδικα θα δουλεύει πάντα;

assert

- Αν μπορούσαμε να ελέγξουμε τις αρχικές/input συνθήκες και τις τελικές/output συνθήκες τότε οτιδήποτε συμβεί θα είναι ένα από τρία ενδεχόμενα:
  - Οι αρχικές συνθήκες είναι λάθος → σημαίνει ότι κάποιο άλλο κομμάτι το προγράμματος είναι λάθος
  - Οι τελικές συνθήκες είναι λάθος → σημαίνει ότι γράψαμε αυτή την συνάρτηση λάθος
  - Η κατανόησή μας για τις αρχικές/τελικές συνθήκες είναι λάθος → σημαίνει ότι δεν έχουμε ξεκαθαρίσει τι πρέπει να κάνουμε
  - (ή τίποτε από τα παραπάνω και όλα είναι εντάξει)
- Αν λοιπόν εμείς μπορούμε να ελέγξουμε αυτά τα τρία πράγματα για κάθε κομμάτι κώδικα τότε αυτό θα δουλεύει πάντα σωστά ή θα αποτύχει κάποιος έλεγχος, χωρίς να υπάρχει άλλο ενδεχόμενο
- Το σκεπτικό λοιπόν των assertions είναι ακριβώς ο έλεγχος των συνθηκών με σαφή τρόπο, ώστε:
  - Να αποτυπώσουμε εμείς τι πιστεύουμε ότι ισχύει για κάθε κομμάτι κώδικα
  - Να κάνει τους σχετικούς ελέγχους το ίδιο το πρόγραμμα κατά την εκτέλεσή του
- Στη C αυτό γίνεται με ένα πολύ απλό module <assert.h> που μας παρέχει μια συνάρτηση (macro για την ακρίβεια)

```
#define assert(condition)  \
  if (!(condition)) printf ("%s:%u: failed assertion `%s'\n", \
                            __FILE__, __LINE__, #condition); exit(...);
```

  - Π.χ. assert (x>0);
  - Αν το x είναι μεγαλύτερο από το 0 τότε η assert δεν κάνει τίποτε και απλά συνεχίζει η εκτέλεση του προγράμματος, διαφορετικά σταματάει η εκτέλεση με ένα μήνυμα ότι η συνθήκη δεν ισχύει στο συγκεκριμένο σημείο του προγράμματος
- Αυτός ο απλός μηχανισμός είναι μια από τις πιο ισχυρές τεχνικές (η πιο ισχυρή) στο να σχεδιάζουμε συστήματα που είναι σωστά

strcpy()

- Ας επιστρέψουμε στο παράδειγμά μας

```
char *strcpy(char *s1, const char *s2) {
        char *s = s1;
        while ( (*(s++) = *(s2++)) != '\0') ;
        return s1;
}
```

- Ποια πρέπει να είναι η αρχική κατάσταση ώστε αυτό το κομμάτι κώδικα να είναι πάντα σωστό;
  - (a) Τα s1, s2 πρέπει να είναι non-NULL (αφού κάνουμε dereference και τους δύο pointers)
  - (b) Το s2 πρέπει να είναι ένα "σωστό" string, αφού προσπαθούμε να το αντιγράψουμε κάπου αλλού (στο s1)
  - (c) Το s1 πρέπει να δείχνει σε "σωστή" και "αρκετή" μνήμη, αφού θα κρατήσει ένα αντίγραφο του s2
- Ποια επιθυμούμε να είναι η κατάσταση του προγράμματος αφού τελειώσει αυτό το κομμάτι κώδικα ώστε να είναι εγγυημένο ότι δεν κάναμε κάτι λάθος;
  - (d) Τα s1,s2 να είναι ίσα (σαν strings)
  - (e) Να μην έχει γίνει καμία άλλη αλλαγή στη μνήμη του προγράμματός μας
- Πως ελέγχουμε αυτές τις συνθήκες;
  - Το (a) είναι μάλλον εύκολο: assert(s != NULL)
    - ή όπως γράφουμε πολλές φορές ιδιωματικά στη C: assert(s);
  - Το (d) είναι επίσης εύκολο: πχ assert(!strcmp(s1, s2));
    - Αν χρειαστεί, γράφουμε εμείς την συνάρτηση strcmp ή όποια άλλη απαιτείται για να εκφράσουμε σωστά την συνθήκη που μας ενδιαφέρει
  - Τι γίνεται με τα (b),(c),(e)?

Έλεγχος κατάστασης μνήμης

- (b) Το s2 πρέπει να είναι ένα ”σωστό“ string, αφού προσπαθούμε να το αντιγράψουμε κάπου αλλού (στο s1)
  - Αυτό σημαίνει ότι θέλουμε το s2 να έχει/δείχνει σε σωστή μνήμη και να τερματίζεται με ‘\0’
  - (b.1) Ο έλεγχος του τελικού ‘\0’ θα μπορούσε να είναι εύκολος σε πρώτη όψη αλλά πως σταματούμε αν δεν υπάρχει ‘\0’;
  - (b.2) Επίσης, τελικά είναι δώρο-άδωρο αν δεν γνωρίζουμε ότι η μνήμη του s2 έχει γίνει allocate με σωστό τρόπο και δεν δείχνει κάπου τυχαία
- (c) Το s1 πρέπει να δείχνει σε “σωστή” και “αρκετή” μνήμη, αφού θα κρατήσει ένα αντίγραφο του s2
  - Αυτή η περίπτωση είναι αντίστοιχη με την (b.2)
  - Αν η C είχε ένα ισχυρότερο type system (strongly typed), όπου τα strings, οι pointers, και η δυναμική μνήμη θα μπορούσαν να ελεγχθούν από τον compiler ότι είναι σωστές μεταβλητές και δείχνουν σε σωστή μνήμη. Άρα μια γλώσσα με ένα ισχυρό type system αφαιρεί από τον σχεδιαστή το βάρος τέτοιων (δύσκολων) ελέγχων.
- (e) Να μην έχει γίνει καμία άλλη αλλαγή στη μνήμη του προγράμματός μας
  - Για αυτή τη συνθήκη θα μπορούσε κανείς να θεωρήσει ότι είναι περιττή, ωστόσο αυτό δεν ισχύει γιατί αν ο κώδικάς μας είναι λάθος, θα μπορούσε καθώς διατρέχουμε τους pointers να αλλάξουμε θέσεις μνήμης που δεν επιτρέπεται
  - Αλλά και ο έλεγχος αυτής της συνθήκης είναι δύσκολος, π.χ. θα έπρεπε να ανιχνεύσουμε writes σε θέσεις μνήμης ή να συγκρίνουμε δύο αντίγραφα της μνήμης πριν και μετά την συνάρτηση, μέθοδοι που είναι πολύ χρονοβόροι, χρειάζονται πολύ μνήμη, ή και τα δύο
  - Βλέπουμε πιθανώς τώρα πόσο σημαντικό θα ήταν το πλεονέκτημα μιας γλώσσας που οι συναρτήσεις δεν μπορούν να αλλάξουν τίποτε άλλο στη μνήμη του προγράμματος εκτός από το να επιστρέψουν μια τιμή (functional γλώσσες). Σε αυτές τις γλώσσες, η συνθήκη αυτή θα ήταν trivially σωστή, πάντα. Αυτό είναι και ένα από τα κύρια επιχειρήματα υπέρ των functional languages (αλλά βέβαια και με πολλά αντεπιχειρήματα).
- Τελικά λοιπόν, και οι τρεις αυτές συνθήκες, ανάγονται στον έλεγχο της κατάστασης της μνήμης του προγράμματος, που δεν μπορεί να γίνει με αυστηρό και πλήρη τρόπο στη C.
  - Σε αυτό το σημείο βλέπουμε αυτή την ισορροπία ανάμεσα σε απόδοση και “ευκολία” που προσπαθεί να πετύχει κάθε γλώσσα και τελικά είναι περισσότερο ή λιγότερο χρήσιμη για συγκεκριμένες δουλειές. Γλώσσες όπως η C που έχουν απλούστερο runtime τείνουν να χρησιμοποιούνται στο σχεδιασμό λογισμικού συστημάτων ενώ γλώσσες όπως η Scala, Java τείνουν να χρησιμοποιούνται για τον σχεδιασμό εφαρμογών.

Assertions

- Η assert είναι απλά ένας μηχανισμός – το βάρος για τους σωστούς ελέγχους πέφτει στον σχεδιαστή του συστήματος.
- Πρέπει να συνειδητοποιήσουμε ότι για να ελέγξουμε τις υποθέσεις χρειάζεται
  - (1) Να τις έχουμε ξεκάθαρες ως προγραμματιστές/σχεδιαστές
  - (2) Να τις εκφράσουμε με κώδικα, ακόμη και αν χρειαστεί να γράψουμε επιπλέον κώδικα από αυτόν που απαιτούν οι λειτουργίες του προγράμματός μας
  - (3) Να εισάγουμε τα κατάλληλα assertions στα σωστά σημεία
- Συνήθως ξεκινούμε με το να προσθέτουμε assertions για
  - input παραμέτρους και return values συναρτήσεων
  - κεντρικές δομές του συστήματός μας όπου ελέγχουμε την συνέπεια των διαφόρων πεδίων, υπο-δομών και λειτουργιών τους
- Ενδεχομένως να χρειαστεί στη διαδικασία αυτή να γράψουμε επιπλέον κώδικα (πέρα από τα assertions) για να πραγματοποιούμε τους ελέγχους που θέλουμε
- Στην πράξη ο κώδικάς μας μπορεί/πρέπει να περιέχει ένα assertion για κάθε μερικές (3-5) γραμμές κώδικα

Παραδείγματα

- Επιλογή από συγκεκριμένες περιπτώσεις

```
switch(what){
            case ONE: ...
            case TWO: ...
            case THREE: ...
            default:
                    assert(0);
}
```

- Ενας server που δεν τελειώνει ποτέ

```
int main(void){
    while (1){
            r = get_next_request();
            process_requrest(r);
    }
    assert(0);
    return 0;
}
```

- Δεν χρειάζεται να ελέγχουμε ” τετριμμένες” περιπτώσεις, π.χ.

```
i = j + 1;
assert (i == j+1);
```

(2) Ελεγχόμενες συνθήκες (Runtime checked errors)

- Τί είναι, σε αντίθεση με τα assertions, ένα runtime-checked error?
- Μια συνθήκη που μπορεί να συμβεί κατά την εκτέλεση του προγράμματός μας και πρέπει το πρόγραμμα να την χειριστεί κατάλληλα, π.χ.

```
p = (char *) malloc(size);
```

→ Πως ελέγχουμε το `p?` με `if (!p) {}` ή με `assert(p)` ?

  - Γενικά, δεν μπορούμε να θεωρήσουμε σε ένα πρόγραμμα ότι πάντα θα βρίσκουμε την μνήμη που θέλουμε
  - Άρα θα πρέπει να ελέγξουμε στο σημείο αυτό το αποτέλεσμα της malloc και να χειριστούμε το error αυτό με τον όποιο τρόπο κρίνουμε κατάλληλο για το πρόγραμμά μας
  - Η αδυναμία της malloc δεν μπορεί να είναι μια συνθήκη που θεωρούμε ότι δεν θα συμβεί ποτέ στο πρόγραμμά μας, με βάση τον σχεδιασμό του (by design)
  - Άρα στην περίπτωση αυτή, χρησιμοποιούμε if () {}:

```
p = (char *) malloc(size);
if (!p) {
        printf("Error out of memory\n");
        return (-1);
}
```

- Άρα τα assertions είναι συνθήκες που με βάση τον σχεδιασμό του προγράμματος δεν μπορούν να συμβούν (εκτός αν υπάρχει κάποιο bug) και επομένως όταν εκτελείται το πρόγραμμά μας (αφού είναι σωστό και το δώσαμε σε τρίτους) δεν χρειάζονται τα assertions
- Τα runtime checked errors, μπορούν να συμβούν κατά την κανονική εκτέλεση και πρέπει να κάνει κάτι για αυτά το σύστημά μας, με κώδικα που είναι μέρος του συστήματος

Ο ρόλος του interface (ενός module/ADT)

- Ας επανέλθουμε στο παράδειγμα της strcmp
- Το αν τα arguments είναι NULL είναι assertion ή runtime-checked error?
  - Και επομένως κατά αντιστοιχία θα κάνουμε τον σχετικό έλεγχο με assert ή if...
- Η απάντηση εξαρτάται από τον ορισμό του interface της συνάρτησης (κατά τον ορισμό του αντίστοιχου module)
  - Αν το interface λέει ότι τα args μπορεί να είναι NULL, τότε αυτό σημαίνει ότι είναι ένα runtime checked error (γιατί πως να αντιγράψουμε NULL strings;) και επομένως το πρόγραμμά μας πρέπει να χρησιμοποιεί if
  - Αν το interface λέει ότι τα args είναι non-NULL strings τότε δεν θα έπρεπε ποτέ να φτάσει να κληθεί η strcmp με NULL arguments και επομένως πρέπει να χρησιμοποιήσουμε assert
- Άρα ένας από τους ρόλους ενός σωστού interface είναι να ορίζει με σαφήνεια τι είναι runtime checked error, ώστε να είναι σαφές πως κάποιος μπορεί να χρησιμοποιήσει αυτό το interface

Development vs. production mode

- Μπορούμε να σκεφτόμαστε ότι τα assertions είναι για να μας βοηθήσουν να σχεδιάσουμε ένα σωστό σύστημα, και αφού το πετύχουμε αυτό δεν χρειάζονται πλέον (γιατί όλα θα δουλεύουν σωστά…)
- Για αυτό τον λόγο, η assert ορίζεται σαν

```
#ifdef DEBUG
#define assert(condition)  \
        if (!(condition)) printf ("%s:%u: failed assertion   `%s'\n", \
                                __FILE__, __LINE__, #condition); exit(...);
#else
        #define assert(condition)
#endif
```

- Οπότε όταν κάνουμε compile με το DEBUG flag να μην είναι ορισμένο, όλα τα assertions θα αφαιρεθούν από το πρόγραμμά μας από τον cpp
- Με βάση αυτό το σκεπτικό, τα assertions δεν πρέπει ποτέ να περιέχουν κώδικα που χρειάζεται για την κανονική εκτέλεση του προγράμματός μας, πχ.

```
assert((--count) != 0);
```

  - Σε αυτή την περίπτωση όταν αφαιρεθεί το assert δεν θα εκτελείται πλέον τον --count. Άρα αν χρειάζεται στο πρόγραμμα, με την αφαίρεση των assertions το συστήμά μας δεν θα δουλεύει πλέον σωστά.

errno.h

- Κάποιες συναρτήσεις στη C επιστρέφουν λάθη όχι μέσω του return value, αλλά μέσω μιας global μεταβλητής errno που δηλώνεται στο errno.h, π.χ.

```
#include <errno.h>
errno = 0;        /* αν γίνει !=0 τότε θα υπάρχει κάποιο λάθος */
f = fopen("file", ...);
if (f == NULL) { /* κάτι δεν έγινε σωστά, αλλά τι; */
        perror("file open returned error: ");
}
```

- Η fopen χρειάζεται να περιγράψει διαφορετικά δυνατά λάθη
- Ωστόσο, η μόνο τιμή λάθους που μπορεί να επιστρέψει είναι το NULL και δεν μπορεί να τα αναπαραστήσει όλα
- Το errno δίνει στην fopen (και σε άλλες συναρτήσεις) την δυνατότητα περιγραφής διαφορετικών λαθών
- Η perror() είναι μια συνάρτηση που ελέγχει την τιμή της μεταβλητής errno που έχει θέσει η fopen και τυπώνει το δικό μας μήνυμα και ένα μήνυμα που σχετίζεται με την τιμή του errno
- Όλες οι δυνατές τιμές του errno και τα αντίστοιχα μηνύματα ορίζονται στο errno.h
- Το errno.h είναι module, και επομένως υπάρχει ένα μόνο αντίγραφο της μεταβλητής errno για όλο το πρόγραμμα κατά την εκτέλεσή του

Reading

- King Ch. 24.{1,2}