HY255 Εργαστήριο Λογισμικού – L22

(a) Exceptions

(b) setjmp/longjmp in C

(c) Signals (signal/raise) in C

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

Error Handling

- Ο κώδικας που χρησιμοποιείται για έλεγχο λαθών σε προγράμματα είναι ”πολύς”, περισσότερος από τον μισό κώδικα ενός προγράμματος (80-20 rule)
- Θα δούμε γιατί συμβαίνει αυτό και τι μπορούμε να κάνουμε ώστε το πρόγραμμά μας να είναι πιο ευανάγνωστο
- Πως αντιμετωπίζουμε errors (runtime checked errors) σε προγράμματά μας;
- Μέχρι τώρα: if (cond) - then – else, π.χ.

```
p = malloc(size);
if (!p) {
    printf(”memory allocation error – not enough memory\n”);
    exit(ENOMEM);
}
```

  - Η χρήση του assert() είναι για να ελέγξουμε design assumptions, όχι για error handling
- Συνήθως σε ένα πραγματικό πρόγραμμα, πολλά λάθη θα πρέπει να τα χειριστούμε με κάποιο τρόπο, αντί απλά να κάνουμε exit()
- Άρα σε πολλές περιπτώσεις στα σημεία που συμβαίνει το error επιστρέφουμε return(some error), αντί για exit()

Ένα απλό παράδειγμα

- Αν το πρόγραμμά μας είναι κάποιος server που επεξεργάζεται requests

```
void server(void){
      while(1){
        r=get_next_request();
        ret = process(r);
      }
      assert(0);
}
```

- Αν γίνει κάποιο λάθος σε ένα request θα επιστρέψει σε κάποιο main loop, θα κάνει κάποιο cleanup και θα περιμένει το επόμενο request

```
void server(void){
      while(1){
        r=get_next_request();
        ret = process(r);
        switch (ret) {
          case ERROR1: cleanup(ERROR1); break;
          case ERROR2: cleanup(ERROR1); break;
          case ERROR3: cleanup(ERROR1); break;
          case success: break;
          default: assert(0);
        }
      }
      assert(0);
}
```

```
int process(Req_t r){
      /* do something */
      if (error1) return(ERROR1);

      /* do something */
      if (error2) return(ERROR2);

      /* do something */
      if (error3) return(ERROR3);

      /* do something */
      return success;
}
```

  - Επομένως, η συνάρτηση process() θα πρέπει να επιστρέφει με διάφορες συνθήκες λάθους που θα συμβούν
- Παρατηρήστε πόσο έχει αυξηθεί ο κώδικας του server στην προσπάθειά μας να χειριστούμε runtime errors
- Παρένθεση: Θα μπορούσε κανείς να σκεφτεί ότι μπορούμε να χειριστούμε τα λάθη στο σημείο που συμβαίνουν ώστε να μην χρειαστεί να επιστρέψουμε στο κεντρικό loop, που θα μείωνε φυσικά το μέγεθος του κώδικα. Δυστυχώς αυτό δεν είναι εφικτό, λόγω των βιβλιοθηκών: Μια βιβλιοθήκη που χρησιμοποιείται από πολλά διαφορετικά προγράμματα δεν μπορεί να χειριστεί το λάθος για όλα τα προγράμματα, μπορεί μόνο να επιστρέψει την σχετική πληροφορία και το κάθε πρόγραμμα να το χειριστεί όπως αυτό κρίνει κατάλληλο για την περίπτωσή του. Θα μπορούσαμε να σκεφτούμε ακόμη πιο exotic λύσεις, πχ να περνούμε σε κάθε library function τον error handling code σαν παράμετρο, αλλά αυτό ουσιαστικά σημαίνει ότι δίνουμε πρόσβαση στην βιβλιοθήκη σε όλο το state του enclosing προγράμματος, που είναι και πολύ πολύπλοκο για τον σχεδιαστή αλλά και ενάντια στο σκεπτικό των βιλιοθηκών.
- This is reality! – and it gets worse...

Function Nesting

- Σκεφτείτε τώρα την περίπτωση όπου το runtime error συμβαίνει ”βαθιά” σε μια nested ακολουθία συναρτήσεων

```
int f1(...){
  /* do something */
  if (error) return E1;
  ret = f2(...);
  switch(ret){
    case E2: /* maybe do some local cleanup */; return E2;
    case E3: /* maybe do some local cleanup */; return E3;
    case success: break;
    default: assert(0);
  }
  return success;
}
int f2(…){
   /* do something */
  if (error) return E2;
  ret = f3(...);
  switch(ret){
    case E3: /* maybe do some local cleanup */; return E3;
    case success: break;
    default: assert(0);
  }
  return success;
}
```

```
int f3(…){
  /* do something */
  if (error) return E3;
  return success;
}

main(){
  while(1){
    r = get_next_request();
    ret = f1(r);
    switch (ret) {
      case E1: cleanup(E1); break;
      case E2: cleanup(E1); break;
      case E3: cleanup(E1); break;
      case success: break;
      default: assert(0);
    }
  }
  assert(0);
}
```

- Ο πιο πολύς κώδικας (κόκκινος) αφορά σε runtime errors
- Επιπλέον, ένα απλό πρόγραμμα έγινε δυσανάγνωστο

Exceptions

- Τα exceptions σε διάφορες γλώσσες είναι ένας μηχανισμός που μπορεί να μας διευκολύνει στο να μειώσουμε κάπως τον κώδικα και να τον κάνουμε πιο ευανάγνωστο – η C ΔΕΝ υποστηρίζει exceptions
- Η βασική παρατήρηση είναι ότι μεγάλο μέρος του κώδικα απλά μεταφέρει τη ροή του προγράμματος από nested σημεία που έγινε το error στο σημείο που θα γίνει ο χειρισμός του
- Τα exceptions είναι ένας μηχανισμός που πετυχαίνουν ακριβώς αυτό: μεταφέρουν την ροή από ένα σημείο σε άλλο με λιγότερο κώδικα
- Ενδεικτική σύνταξη: try-catch-raise

```
try {
     /* regular code */
}
catch (E1){
     /* error handling */
}
catch (E2){
     /* error handling */
}
catch (E3){
     /* error handling */
}
```

- Όποια συνάρτηση συναντήσει κάποιο runtime error μπορεί να κάνει issue

```
raise E;
```

- Τα exceptions έχουν συνήθως κάποιον ειδικό τύπο που ορίζεται από την γλώσσα και ένα πρόγραμμα μπορεί να ορίζει exceptions κατά βούληση, όπως και τις μεταβλητές/σταθερές

Function Nesting (revisited)

- Το παράδειγμα με τις nested συναρτήσεις γίνεται πλέον

```
int f1(...){
  /* do something */
  if (error) raise E1;
  ret = f2(...);
  return success;
}
int f2(…){
  /* do something */
  if (error) raise E2;
  ret = f3(...);
  return success;
}
int f3(…){
  /* do something */
  if (error) raise E3;
  return success;
}
```

```
main(){
  while(1){
     try {
       r = get_next_request();
       ret = f1(r);
     }
     catch (E1) {cleanup(E1); break;}
     catch (E3) {cleanup(E1); break;}
     catch (E3) {cleanup(E1); break;}
  }
  assert(0);
}
```

- Εξακολουθεί να περιέχει κώδικα για τον χειρισμό των runtime errors
- Αλλά τουλάχιστον έχουμε “γλιτώσει” από όλα τα ενδιάμεσα βήματα και κώδικα
- Και σίγουρα ο κώδικας είναι πιο ”αναγνώσιμος”
- Παρατήρηση: Τα exceptions δεν “εξαφανίζουν” το πρόβλημα του χειρισμού λαθών. Στο πρόγραμμά μας πρέπει να σχεδιάσουμε/σκεφτούμε πως θα χειριστούμε λάθη και να γράψουμε σχετικό κώδικα. Απλά, κάνουν αυτόν τον κώδικα λίγο πιο απλό σε θέματα “διαδικαστικά” (όχι σε θέματα ουσίας).

Που δεν χρησιμοποιούμε exceptions

- Τα exceptions προορίζονται για να χειριστούμε runtime errors, όχι οποιαδήποτε συνθήκη στο πρόγραμμά μας
- Π.χ. το EOF είναι ένας έλεγχος που κάνουμε τακτικά

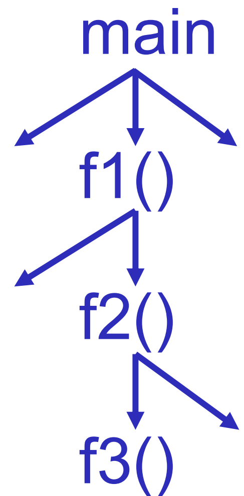
```
while ((c = getchar()) != EOF) {
      putchar(c);
}
```

- Θα μπορούσαμε να γράψουμε

```
try {
      while (1) {
            c = getchar();
            putchar(c);
      }
} catch(EOF){
      /* end of file - do what is next */
}
```

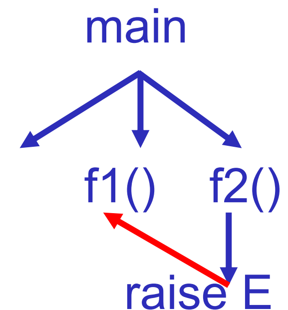
- Και η συνάρτηση getchar() θα κάνει `raise EOF`
- Αλλά το EOF δεν είναι λάθος, είναι απλά μια συνθήκη
- Αν γράψουμε τον κώδικά μας έτσι, θα είναι σαν να λέμε σε κάποιον τρίτο ότι ο κώδικας που βρίσκεται στα catch clauses προορίζεται για runtime errors που είναι το λάθος ”μήνυμα”

Πως υλοποιείται το try-catch-raise?



- Έστω ένας call graph κάποιου προγράμματος
- Όταν κάνουμε raise στην f3(), "πηγαίνουμε" στο σημείο που υπάρχει το try-catch για το συγκεκριμένο exception, π.χ. στη main. **Πως;**
- Μπορούμε να το πετύχουμε αυτό στη C με κάποιον από τους μηχανισμούς που γνωρίζουμε;
  - Αλλαγή ροής στη C γίνεται με
  - if-then-else / switch
  - for/while/etc
  - break - continue
  - goto
  - Τι κάνει κάθε ένας από αυτούς τους μηχανισμούς;
- Μπορούμε λοιπόν να χρησιμοποιήσουμε κάποιον από αυτούς τους μηχανισμούς για να υλοποιήσουμε exceptions;
- **Όχι**
  - Όλοι αυτοί οι τρόποι αλλαγής ροής έχουν διάφορους (και διαφορετικούς) περιορισμούς αλλά και έναν κοινό περιορισμό
  - Όλοι τους μεταφέρουν την ροή μέσα στην ίδια συνάρτηση
  - Κανείς τους δεν μπορεί να μεταφέρει την ροή έξω από ή μέσα σε μια άλλη συνάρτηση
- Τα exceptions **απαιτούν** κάποιο μηχανισμό που να μεταφέρει την ροή σε μια άλλη συνάρτηση, οπότε αυτός ο μηχανισμός πρέπει να "τακτοποιεί" και την στοίβα, ώστε όταν μεταφερθεί ο έλεγχος η target συνάρτηση να χρησιμοποιεί το σωστό stack frame στη στοίβα
- Στην υλοποίησή του το try-catch "θυμάται" το stack frame της συνάρτησης που περιέχει τον κώδικα του try-catch και όταν γίνει το raise, επιστρέφει την ροή στο σωστό stack frame και στη σωστή εντολή (αμέσως μετά το try όπου ελέγχονται όλα τα catch clauses)
- Μπορούμε άραγε να χρησιμοποιήσουμε την try-catch-raise σε οποιεσδήποτε συναρτήσεις, π.χ.:

```
main(){      f1(){              f2(){
 f1();        try{...}           ...
 f2();        catch(E1){...}     raise E1;
}            }                  }
```



- Όχι, γιατί η κλήση της f2() που γίνεται το raise δεν είναι nested μέσα στην f1 και επομένως δεν υπάρχει stack frame της f1 στη στοίβα τη στιγμή του raise

Στη C

- Η C <u>δεν</u> παρέχει exceptions
- Διαθέτει ωστόσο δύο άλλους παρόμοιους μηχανισμούς, που μπορούν να χρησιμοποιηθούν για να υλοποιήσουν (κάποια μορφή) exceptions
- (a) setjmp/longjmp
- (b) signal/raise

setjmp/longjmp

- Η stdlib της C διαθέτει το module <setjmp.h>

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

- Η setjmp() σημειώνει στην μεταβλητή env το σημείο στο οποίο καλείται η setjmp και μπορεί αργότερα να επιστρέψει σε αυτό το σημείο από άλλες συναρτήσεις με την χρήση της longjmp
- Η longjmp(env, val) αλλάζει την ροή στο σημείο που υποδεικνύει η μεταβλητή env και "περνάει" την τιμή val σαν τιμή επιστροφής της setjmp()
- Μετά την longjmp θα εκτελεστεί ότι ακολουθεί την setjmp()
- Άρα η setjmp είναι μια συνάρτηση που θα κληθεί μια φορά, αλλά μπορεί να επιστρέψει πολλές φορές
- Για να διακρίνουμε αυτές τις κλήσεις, η setjmp επιστρέφει 0 όταν επιστρέφει από δική της κλήση, και την τιμή val όταν επιστρέφει μετά από μια κλήση longjmp(env,val)

Παράδειγμα

```
#include <setjmp.h>
#include <stdio.h>

static jmp_buf env;

void f1(void);
void f2(void);

int main(void){
  int ret;
  ret = setjmp(env);
  printf("setjmp returned %d\n", ret);
  if (ret!=0){
    printf("terminate from longjmp\n");
    return 0;
  }
  f1();
  printf("terminated normally\n");
  return 0;
}

void f1(void){
  printf("f1 begin\n");
  f2();
  printf("f1 end\n");
}

void f2(void){
  printf("f2 begin\n");
  longjmp(env, 1);
  printf("f2 end\n");
}
```

Output:

```
setjmp returned 0
f1 begin
f2 begin
setjmp returned 1
terminate from longjmp
```

Signals in C

- Τα signals είναι ο μηχανισμός της C (σε συνεργασία με το λειτουργικό σύστημα – σχεδιάστηκαν μαζί με το unix) για να επικοινωνούν διαφορετικά προγράμματα (processes) ή το λειτουργικός σύστημα με ένα πρόγραμμα
- Ωστόσο, τα signals μπορούν να χρησιμοποιηθούν και ”εσωτερικά” σε ένα πρόγραμμα
- Ένα πρόγραμμα μπορεί να ορίσει (με τη συνάρτηση signal) ότι όταν ”σταλεί“ ένα signal θα τρέχει ένας handler (μια συνάρτηση) του προγράμματος
  - Prototype: `void (*signal(int sig, void (*func)(int)))(int);`

```
#include <signal.h>
void handle_sigint(int context){
    printf("caught SIGINT");
}
signal(SIGINT, handle_sigint);
```

- Ένα πρόγραμμα μπορεί να ”στείλει” ένα signal με τη συνάρτηση raise()

  Prototype: `int raise(int sig);`

```
raise(SIGINT);
```

- H C ορίζει μερικά (predefined) signals που μπορούν να σταλούν από το ίδιο το πρόγραμμα αλλά και από άλλους (πχ από το λειτουργικό σύστημα προς το πρόγραμμα)
  - SIGSEGV, SIGINT, SIGFPE, …
- Στη C τα signals είναι “global” μπορούν να συμβούν οπουδήποτε (raise) και θα κληθή πάντα η ίδια συνάρτηση (που έχει οριστεί για το συγκεκριμένο signal)

Reading

- King 24.{3,4}
- setjmp/longjmp
  - Section 2.8 of (online) The C Library Reference Guide, Eric Huss, 1997. [html]
  - Section 9.7 of (online) The C Book, Mike Banahan, Declan Brady and Mark Doran, Second Edition, 2003. [html] [pdf (updated 2020)]
- signals
  - Section 2.9 of (online) The C Library Reference Guide, Eric Huss, 1997. [html]
  - Section 9.8 of (online) The C Book, Mike Banahan, Declan Brady and Mark Doran, Second Edition, 2003. [html] [pdf (updated 2020)]