

# Alcuni errori tipici

Frammento 1

# Sulla gestione errori...

- Deve essere controllato l'esito delle funzioni di libreria

- malloc-calloc non controllate!
- Spesso si controlla il puntatore DOPO averlo dereferenziato come in

```
scheda_t * res  
malloc(sizeof(scheda_t));  
res->autore = ...  
if ( res == NULL) .../*gestione errore  
*/
```

# Sulla gestione errori (2)

- Mai fidarsi del valore dei parametri passati a una f.ne di libreria
  - Se una funzione puo' essere usata da altri programmi non noti testare sempre se i valori sono consistenti con quelli che ci aspettiamo (if, assert etc...)
  - Si doveva fare per le funzioni nell'interfaccia bib.h
- Le funzioni che modificano errno devono documentare la cosa
  - Nel commenti in bib.h e' gia' documentato
  - Dovete aggiungere commenti se usate piu' codici di errori

# Sulla gestione errori... (3)

- Errore viene interpretata da perror
  - è necessario assegnare a errno solo valori noti (ENOMEM, EINTR etc..)
- Le stampe di messaggi di errori devono essere effettuate su stderr
  - perror(...) != fprintf(stderr, ....)
- è preferibile non fare stampe di errori o di debug dentro le funzioni di libreria ma ritornare un valore (setando errno opportunamente)
  - come accade per le librerie che usiamo di solito

# Sulla gestione errori... (4)

- è VIETATISSIMO chiamare la `exit()` o la `_exit()` da dentro le funzioni di libreria

# Memoria

- Se si effettuano più allocazioni in sequenza
  - è necessario liberare tutta la memoria precedentemente allocata se una allocazione va male
- è preferibile effettuare malloc separate per diverse variabili e free separate delle stesse
  - accedere a indirizzi dinamici non generati da malloc direttamente può generare problemi di portabilità e complica la leggibilità del programma
- dove possibile è sempre meglio usare variabili locali piuttosto che dinamiche
  - più efficiente, non ‘genera’ memory leak

# Commenti

- Non devono ripetere passo passo quello che è evidente dal codice!
- è preferibile:
  - scrivere un commento esauriente all'inizio della funzione, con eventuali note algoritmiche
  - commentare sinteticamente all'inizio di un blocco di 5-15 statement per spiegare cosa sta per succedere
  - commentare bene le variabili globali (inclusa la politica di modifica)
  - commentare brevemente le var locali dal significato non ovvio

## Commenti (2)

```
int scompatta (char*a, scheda_t * ...) {  
int i = -1;  
int j = -1;  
int k = -1;  
int w = -1;  
int d = -1;  
int c = -1;  
int l = -1;  
int p = -1;  
int x = -1;  
.....  
/* seguono centinaia di linee di codice che  
   usano le variabili in tutte le possibili  
   combinazioni */  
}
```

# Miscellanea

- è *PESSIMA norma* usare costanti numeriche o caratteri cablate nel codice

Es: `write(4, buf, 29), malloc(28),  
open("/tmp/cicciopippo")`

Questo rende difficile leggere e mantenere il codice (28 e 29 sono legati fra di loro o scollegati ? Quel'e' il loro uso ? )

Bisogna fattorizzare le costanti intere e stringa con opportune `#define`, dando loro nomi significativi ed usare tali nomi consistentemente nel codice per esplicitare i legami

Es: `#define LUNG 30`

`write(4, buf, LUNG-1), malloc(LUNG-2),`

# Miscellanea (2)

- *Quasi nessuno ha usato le costatnti del tipo enumerato campo\_t !*

```
Es: switch(campo) {
  case 0:{
    ...
    break;}
  case 1:{
    ...
    break;}
  /* Etc ... INVECE DI ... */
  switch(campo) {
  case AUTORE:{
    ...
    break;}
  case TITOLO:{
    ...
    break;}
  Etc ...
```

# Miscellanea

- *Quasi nessuno ha usato le costanti del tipo enumerato campo\_t !*
  - Qualcuno si e' addirittura rifatto delle define per ridefinire tutti i valori di nuovo

```
#define CAUTORE 0
```

```
#define CTITOLO 1
```

```
...
```

# Miscellanea

- È buona norma dichiarare static le funzioni private (che servono solo in un file)
- Non si devono usare caratteristiche non standard ISO
  - mixed decl and code, array dinamici, funzioni annidate
- Leggere sempre bene il man
  - e non assumere niente che non sia scritto lì
  - es: `strcmp(s,c)` non ritorna -1 se  $s < c$  ma può tornare un numero negativo arbitrario ...

# Miscellanea

- Molte funzioni di parsing sono lunghe e contorte
  - Qualcuno ha definito tre funzioni diverse per contare tre caratteri diversi (bastava un parametro)
  - Molti hanno usato la `strtok()` per fare veramente di tutto di piu'
    - Per alcune cose si potevano usare altre funzioni piu' semplici e l'aritmetica dei puntatori
- In molti casi sono state ri-implementate funzioni di libreria es:
  - esiste la **`isdigit()`** per capire se una carattere e' una cifra 0—9
  - La **`strtol()`** o la **`atoi()`** possono essere usate per convertire da stringa ad intero
  - Nella `printf()` placeholder **`%d`** puo' essere modificato in **`%2d`** per stampare solo due cifre di un intero
  - **`strptime()`** e **`strftime()`** possono essere usate per codificare da/verso il formato della data del prestito

# Miscellanea

- Funzioni grosse e contorte o troppo piccole e frammentate rendono il codice difficile da leggere e da mantenere
  - Bisogna FATTORIZZARE il codice replicato !
  - Es: Pochissimi si sono accorti che la `print_scheda` e' una store records sullo `stdout` ed hanno duplicato il codice
  - Molti hanno duplicato il codice di parsing su tutti i nove campi
  - Alcuni hanno implementato 9 volte il bubblesort, una per ogni campo
  - Alcuni hanno liste lunghissime di free che si ripetono uguali prima di ogni return, dovute anche ad un uso sconsiderato dell'allocazione sullo heap (var dinamiche)

# Miscellanea

- Alcuni condividono globali fra una funzione e l'altra es:

```
int stampa;
```

```
void print_scheda (...){
```

```
...
```

```
  stampa=-1;
```

```
}
```

```
Int store_records (.....) {
```

```
.....
```

```
  stampa = 0;
```

```
  if (stampa == 0) ...
```

```
}
```

# Miscellanea

- Alcuni condividono globali fra una funzione e l'altra es:
  - L'uso di globali comporta codice difficile da leggere e da mantenere, e crea dipendenze difficili da scovare
  - La situazione peggiora in presenza di piu' thread
  - Le variabili globali devono essere evitate a meno di casi di assoluta necessita'
  - È una pessima idea ad esempio usare delle globali per gli indici dei cicli ed usarle in funzioni diverse

# Miscellanea

- Meglio `strtol()` che `atoi()` nelle conversioni
  - Se no non si riesce a gestire gli errori
- Molti di coloro che hanno scritto i commenti doxygen non si sono preoccupati di compilare ed andare a gestire i warning!
- Nel make spesso sono state dimenticato le dipendenze da include (usate **gcc -MM !!!**)

# Miscellanea

- È VIETATISSIMO includere i .c in altri .c
  - Questo crea duplicazione di codice nei .o e contravviene a tutte le norme di organizzazione del codice C
- È anche da evitare l'inclusione di .h in altri .h
  - Questo permette di evitare dipendenze non necessarie e nascoste in innocui include
- Nell'accesso degli elementi di un array è sempre da preferire la notazione `a[i]` alla `*(a+i)`
  - Sono equivalenti ma la prima rende il codice più leggibile e manutenibile

# Miscellanea

- **Attenzione agli accessi alla memoria già deallocata** Es

...

```
del = p;
```

```
free (del);
```

```
p = p -> next ;
```

....

- **Attenzione agli accessi a variabili non inizializzate**
  - Spesso va bene perché la memoria è stata azzerata ma possono generare errori in contesti diversi
  - Fatevi aiutare da valgrind