# Alcuni errori tipici

Frammento 1

# Sulla gestione errori...

- Deve essere controllato l'esito delle funzioni di libreria
  - malloc-calloc non controllate!
- Mai fidarsi del valore dei parametri passati a una f.ne di libreria
  - testare sempre se i valori sono consistenti con quelli che ci aspettiamo (if, assert etc…)
- Le funzioni che modificano errno devono documentare la cosa
  - modello man, basta nel commento all'inizio

# Sulla gestione errori… (2)

- Errno 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 (settando errno opportunamente)
  - come accade per le librerie che usiamo di solito

# Sulla gestione errori… (3)

- Molti non fanno *rollback* nella new_smat()
  - Se fallisce una delle allocazioni si esce senza deallocare le parti già allocate sullo heap!

- 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 una 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

# 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 manutenere 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

- Leggere sempre bene il man
  - e non assumere niente che non sia scritto li
  - es: memcpy(NULL, …) segfault su molti sistemi
- Funzioni grosse e contorte o troppo piccole e frammentate rendono il codice difficile da leggere e da manutenere
  - Bisogna FATTORIZZARE il codice replicato !
  - Bisogna definire funzioni di appoggio dove serve
- la ricorsione per molti è ancora una bestia strana
- Le funzioni private (che servono solo in un file) devono essere dichiarate *static*

# Miscellanea

- Meglio strtod() che atof() 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!
- Alcuni hanno warning gravi se compiliamo con –Wall –pedantic ....
  - Tutti i warning gravi corrispondono ad errori potenziali e vanno gestiti

# Miscellanea

- É 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 VIETATA l'inclusione di .h in altri .h
  - Anche questo crea duplicazione e genera errori difficili da trovare
  - Non sarà tollerato nel progetto finale
  - **`sparse.c`** doveva includere **`sparse.h`** !