Defensive Programming

Tecniche generali
Robustezza
Invarianti ed asserzioni

Robustezza

- Metodologie,
  Lezione 7 – Robustezza
  - Tutto

Defensive programming 1/2

- Parte delle tecniche viste nella precedente slide sono un sottoinsieme anche delle tecniche di *bug prevention* e *defensive programming*
- In quanto al bug prevention, leggiamo la sezione Prevention in
  - http://en.wikipedia.org/wiki/Software_bug
  - Nella stessa sezione, seguire il link Programming Style (tralasciare poi la sottosezione Lists)

Defensive programming 2/2

- http://en.wikipedia.org/wiki/Defensive_programming
  - Introduzione e sezioni
    - Secure programming
    - Some defensive programming techniques
      - tranne le sottosezioni The legacy problems, Secure Input / Output Handling, Canonicalization, Principle of least privilege
      - Infine nella sottosezione Other techniques, tralasciare Design By Contract ed Exceptions

Invarianti

- Seguono delle slide in inglese sull'uso degli invarianti e della macro assert
  - Le slide sono prese da “Thinking in C++”, Volume 2

Defensive programming

- Writing perfect software may be virtually impossible
- Bug (Fault) -> Error state -> Failure
- **Bohr Bugs** and **Mandel Bugs**
  - Usually easy to add and hard to find
- But a few defensive techniques, routinely applied, can definitely help narrow the gap between code and ideal

Invariants 1/2

- Code is an expression of an attempt to solve a problem
- It should be clear to the reader what the programmer was thinking of when he/she designed a given piece of code
- At certain points in the program the programmer should be able to make *bold statements* on some condition to hold
  - If he/she cannot, he/she did not yet really solve the problem

Invariants 2/2

- Invariant:

  A statement that is invariably true at the point where it appears in the program

Esercizio

- Aggiungere tutti gli invarianti possibili al progetto
- Confrontare quello che si è riusciti a fare con *progetto_1file_inv/GSeq_1file_inv.cc*

Translating invariants 1/2

- Sometimes an invariant is expressed in a higher level mathematically-based language
- It is hard to translate it into an algorithmic form and express it using the same language as the program
- A certain amount of cleverness is needed to translate the ideal of what we would like to assert into something that is algorithmically feasible to check

Checking invariant violation

- Sometimes the condition depends on user input
  - Impossible to always prevent invariant violation
- Most often the invariant depends only on the code
  - It always holds if the design is correctly implemented
  - And ...

Assertions

- It is clearer to make an assertion

  Positive statement, expressed using the same language as the program, that reveals the design decisions/assumptions

C *assert*

- The C standard Library provides the *assert(expr)* macro (presented in *assert.h* and *cassert*)
  - If expr *!= 0* the execution continues uninterrupted
  - Otherwise:
    1. A message containing the offending expression is printed along with its source file name and line number
    2. The program aborts

Esercizio

- Ovunque possibile, verificare il rispetto degli invarianti mediante asserzioni
- Confrontare quello che si è riusciti a fare con *progetto_1file_ass/GSeq_1file_ass.cc*

Puntatori a funzione 1/2

- Nella prossima slide si farà riferimento al programma *yfq-mod-noop-minimal.cc*, che contiene la seguente definizione:
  `void (*fun_array[8])() = { ... } ;`
- Senza entrare in dettagli, *fun_array* è un array di 8 puntatori a funzione, ove ciascuna funzione ha valore di ritorno void e nessun argomento
- Gli 8 puntatori sono inizializzati con l'indirizzo di 8 funzioni diverse

Puntatori a funzione 2/2

- Ad ogni iterazione del ciclo che segue tale definizione, si genera un indice casuale *i* da 0 a 7
- Mediante tale indice, si invoca casualmente la funzione puntata dall'elemento *i*-esimo dell'array di puntatori a funzione
- Per invocare tale funzione, si utilizza la sintassi:

```
fun_array[i]() ;
```

Why should I waste my time

- ... adding boring invariants and assertions?
- *yfq-mod-noop-minimal.cc*
- After reading the following slides, execute and find the bug

Originale e semplificazioni 1/2

- Il programma *yfq-mod-noop-minimal.cc* è una versione 'finta' ed abbreviata di un modulo del kernel Linux che si occupa di accesso al disco
  - La maggior parte delle istruzioni originali sono state trasformate in semplici stampe delle istruzioni che andavano eseguite
- Nella versione originale alcune funzioni del modulo potevano essere erroneamente invocate mentre un'altra delle funzioni del modulo stesso erano in esecuzione

Originale e semplificazioni 2/2

- Per controllare che questo non accada, alcune funzioni sono corredate dall'invocazione di una funzione *inside* ed una funzione *outside*, rispettivamente, all'inizio ed alla fine delle funzioni stesse
  - Tramite queste funzioni si può controllare se questo accade e fermare in tal caso il sistema

Why should I waste my time

- Compile *yfq-mod-noop-minimal.cc*, execute it and find the bug
- Bohr or Mandel bug?
- What about invariants?

Osservazioni 1/2

- Ad ogni esecuzione, la sequenza di chiamate di funzione prima del crash è sempre la stessa?
  - No, quindi Mandel bug
- Per quale valore di *entered* il programma fallisce?
  - Minore di -1, quindi prima del fallimento può avvenire corruzione della memoria e pertanto ulteriore casualità di comportamento del programma stesso
- Come sarebbe andata se avessimo controllato controllato l'invariante sulla variabile *entered*?

Osservazioni 2/2

- Come sarebbe andata se avessimo banalmente controllato l'invariante sulla variabile *entered*?
- Aggiungere il controllo ci sarebbe costato qualche minuto di tempo, e ci avrebbe risparmiato un lungo e spiacevole lavoro di debugging

NDEBUG

- It is possible to turn on and off assertions
- Turn off:
  #define NDEBUG
- Turn on:
  #undefine NDEBUG
- *<cassert>* must be (re-)included after (un)defining *NDEBUG*
- A common way to define a macro for an entire program is as a compiler/make option

Esempio

- Un esempio di uso della macro *NDEBUG* per disattivare le *assert* è riportato sempre in *progetto_1file_ass/GSeq_1file_ass.cc*

Osservazione 1/2

- Usare invarianti ed asserzioni ha due caratteristiche molto positive:
  ١) E' semplice e pratico
  ٢) E' un approccio sempre vincente:
    ١) Se nessuna *assert* fallisce siamo più confidenti nella correttezza del codice
    ٢) Se una *assert* fallisce:
      ١) Se c'è un errore nel codice lo troviamo e lo correggiamo
      ٢) Se non ci sono errori nel codice, è sbagliato l'invariante

Osservazione 2/2

- Nell'ultimo sotto-caso, se l'invariante sbagliato è una conseguenza di un modello errato di quello che fa il codice, si ha l'opportunità di capire l'errore concettuale e correggere il modello
- In conclusione, qualsiasi caso porta ad un risultato positivo

Procediamo con il progetto

- Cercando di applicare al meglio quanto visto finora sulla qualità del software, la documentazione ed il defensive programming
- Facciamo un altro passo nello sviluppo del progetto
  - Slide 14 e seguenti di *prog-II-progetto*