Debugging

Kernighan & Pike Cap. 5

Debugging (1)

- Fin ora abbiamo sempre ragionato come se tutto funzionasse sempre al primo tentativo.
- Ovviamente non è sempre così.
- I programmatori esperti sanno che dovranno spendere tanto tempo nella programmazione quanto nel debugging.
- La parola debugging viene dall'inglese bug (baco).
- Alle fasi di progettazione e programmazione segue la fase di testing e, se -come spesso accade- il programma non funziona come dovrebbe, è necessario trovare l'errore. Questo è lo scopo della fase di debugging.

Debugging (2)

La fase di debugging può richiedere una quantità di tempo imprevedibile, quindi si deve fare di tutto per semplificare questa attività, seguendo le regole di buona progettazione, tenendo un buono stile di programmazione, verificando i casi limite, mantenendo limitato il numero delle variabili globali.

Debugging (3)

Il C ha alcune caratteristiche implicite che sono soggette ad errori:

- Conversione automatica dei tipi
- Potenzialità dei puntatori
- Mancato controllo delle dimensioni dei vettori
- …

I programmatori dovrebbero conoscere i potenziali "pericoli" per evitarli.

N.B. Nessun linguaggio è in grado di evitare che il programmatore commetta errori!

Strumenti di debugging

I più noti compilatori hanno di solito uno strumento di debugging (debugger), all'interno dell'ambiente di sviluppo, che integra la fase di scrittura, di compilazione e di esecuzione del programma.

Il debugger contiene, di solito, un'interfaccia grafica per:

- L'analisi del codice riga per riga o funzione per funzione
- L'introduzione di breakpoints
- La visualizzazione del contenuto delle variabili

Se non si può (o non si sa) usare il debugger, è necessario avere altri strumenti per risolvere i problemi, man mano che si presentano…

Cercare gli indizi (1)

La maggior parte degli errori sono semplici e si possono trovare con tecniche elementari.

Eccone alcune.

1. **Esaminare l'output errato e procedere a ritroso**: riflettere per tentare di capire cosa sia potuto succedere. Per capire dove si sia verificato l'errore, si possono inserire delle stampe di controllo, che aiutino a tracciare l'andamento del programma.

Cercare gli indizi (2)

2. **Riconoscere errori familiari**
   spesso gli errori si ripetono. Cercare di memorizzare a cosa corrisponde un certo errore aiuta ad identificarlo subito, qualora esso si riproponga.
   Alcuni errori ricorrenti sono:
   - Tentativo di accesso scorretto alla memoria
     `int n;`
     `scanf("%d", n);` (corretto: `scanf("%d", &n);`)
   - Tipi non compatibili e conversioni errate
     `int n=1; double d=4.0;`
     `printf("%d %f", d, n);` (corretto: `printf("%f %d", d, n);`)
   - Dimenticare di inizializzare una variabile locale
     la memoria allocata dalle variabili locali, e dalla malloc è "sporca", e il suo utilizzo dà risultati imprevedibili.

Cercare gli indizi (3)

3. **Esaminare la modifica più recente**
   se si tenta di costruire il codice in modo da avere uno scheletro funzionante, e poi si aggiungono parti ben definite, la comparsa di un errore indicherà anche la parte a cui è dovuto.
   BUONA REGOLA: conservare sempre la versione precedente funzionante

4. **Leggere prima di digitare**
   capita spesso di non sapere esattamente cos'è che non va, e di procedere a tentoni, provando a modificare parti che non hanno nulla a che fare con l'errore.
   BUONA REGOLA: resistere alla tentazione di scrivere: pensare è un'utile alternativa!

Cercare gli indizi (4)

5. **Spiegare il proprio codice a qualcun altro**
   spiegare il proprio codice ad un'altra persona significa spesso spiegare l'errore a se stessi.

6. **Eseguire il debugging subito, non dopo**
   se vi accorgete che qualcosa non va, cercate di risolvere subito, senza rimandare a quando avrete sistemato cose più urgenti, il rischio è che vi dimentichiate dell'errore, o che esso non si riproduca apparentemente più, e che riesca fuori quando meno ve lo aspettate.

Se non ci sono indizi…(1)

A volte non si ha nessuna idea di cosa stia succedendo. In tal caso, si possono provare alcune strade.

1. **Rendere riproducibile l'errore**:
   è molto difficile rincorrere un errore che non si verifica sistematicamente, quindi il primo passo da fare è quello di capire come riprodurlo.
   Se non si riesce a riprodurre l'errore tutte le volte, è necessario capirne il motivo: se c'è un insieme di condizioni che lo governa, probabilmente le variabili interessate in quelle condizioni ne hanno qualcosa a che fare.
   In ogni caso, essere in grado di riprodurlo, significa diminuire i tempi d'attesa e, quindi, risolverlo più in fretta.

Se non ci sono indizi…(2)

2. **Rimpicciolire il problema**:
   è buona norma tentare di rendere il più piccolo possibile l'input per cui il problema si verifica: questo diminuisce la dimensione del problema ed, in ogni caso, abbassa i tempi di inserimento dell'input.
   A volte si può procedere tramite una ricerca binaria: si elimina metà dell'input e si osserva se l'errore c'è ancora; se si, si va avanti, altrimenti si torna alla metà eliminata.

Se non ci sono indizi…(3)

3. **Visualizzare del testo per circoscrivere la ricerca**
   le stringhe di testo non vanno inserite a casaccio, ma seguendo dei criteri:
   - E' bene usare frasi significative (ad esempio: "non può arrivare qui!"),
   - Non usare mai due frasi uguali, in modo da sapere esattamente quale comando abbia prodotto ciascuna stampa
   - Stampare i valori delle variabili via via che il programma viene eseguito ed osservare se i valori sono corretti o, almeno, possibili (ad esempio, non è ragionevole che un puntatore sia un numero negativo o molto piccolo)

Se non ci sono indizi…(4)

4. **Tenere un archivio**
   se la ricerca di un errore prosegue per un po' di tempo si rischia di replicare i tentativi già fatti.
   BUONA REGOLA: mantenere traccia dei tentativi e dei rispettivi risultati.

5. **Banalità**
   a volte l'errore è più banale di quello che si pensi: può essere un errore dovuto al calcolo erroneo delle precedenze degli operatori da parte del programmatore oppure un punto e virgola al posto sbagliato, un = trasformato in == o viceversa, argomenti di una funzione dati in un ordine errato…
   BUONA REGOLA: ricontrollare anche i dettagli

Se non ci sono indizi…(5)

6. **Debugger "manuale"**
   è spesso utile simulare il calcolatore a mano: provate ad eseguire le istruzioni una per una tenendo traccia del contenuto della memoria, cercando di verificare mentalmente se quello che sta accadendo è proprio quello che deve accadere...

7. **L'ultima spiaggia**
   se, dopo aver cercato a lungo, non siete riusciti a trovare l'errore, prendetevi una pausa e fate qualcosa di diverso per un po'. Se neanche questo funziona, chiedete a qualcun altro di dare un'occhiata al vostro programma: è facile che la vostra mente non sia più in grado di scorgere quello che è sotto i vostri occhi.

Errori non riproducibili

- Gli errori non persistenti sono i più difficili da affrontare.
- Il fatto che il comportamento sia non deterministico rappresenta già un'informazione: significa che il problema non risiede nell'algoritmo, ma che in qualche modo il programma usa informazioni che cambiano ogni volta che il programma viene eseguito.
  - Avete inizializzato tutte le variabili?
  - Siete sicuri di non aver scritto al di fuori della memoria allocata?
  - Avete rilasciato con free tutta la memoria che non utilizzate più?
  - Siete sicuri di non allocare o deallocare la stessa memoria più di una volta?

Alcuni strumenti (1) DD p.508

- Il costrutto condizionale del preprocessore consente al programmatore di commentare larghe porzioni di codice, per evitare che siano compilati.
- Nel caso in cui (sperabilmente sempre!) il codice contenga dei commenti, `/*` e `*/` non possono essere usati, ma è possibile usare il seguente costrutto del preprocessore:

```
#if 0
  codice che non deve essere compilato
#endif
```

Per consentire la compilazione lo 0 deve essere sostituito con 1

Alcuni strumenti (2)

- Se si vuole evitare di rimuovere tutti i printf con le stringhe di debugging, è possibile usare il seguente costrutto:

```
#ifdef DEBUG
  printf("stringa da stampare");
#endif
```

- Questa istruzione farà sì che la printf venga eseguita solo se è definita all'inizio del programma una costante DEBUG (#define DEBUG). Se la direttiva #define viene rimossa, la printf verrà ignorata.
- Nei programmi corposi è consigliabile usare a questo scopo diverse costanti simboliche, in modo da isolare le stampe a blocchi.

Conclusione

- Affrontata correttamente, la fase di debugging può essere anche divertente, come risolovere dei problemi enigmistici.
- Che ci si diverta o no, il debugging è un'attività che va praticata regolarmente.
- Alla fine della stesura di un programma sarebbe bello non trovare errori, perciò cercate di evitarli scrivendo il codice nel miglior modo possibile e, prima ancora, progettando opportunamente l'algoritmo.

Un caso di studio: manipolazione dei reali in C (1)

- I numeri reali sono rappresentati in binario nel computer.
- La rappresentazione binaria di alcuni numeri reali, che ci sembrerebbero perfettamente rappresentabili, crea in realtà dei problemi.
- Ragioniamo in base 10: 1/3=0.333…
  - i numeri reali sono memorizzati con un numero limitato di byte, e quindi non sarà mai possibile esprimere con esattezza numeri periodici
  - come vedremo nel seguito, il problema non riguarda solo i numeri periodici…

Un caso di studio: manipolazione dei reali in C (2)

- Sappiamo che un reale (ad es. un double) ha a disposizione un numero finito di bytes per essere rappresentato (ad es. 8)
- Quanti numeri è possibile rappresentare con 8 bytes?
  - 8 bytes=64 bits quindi posso rappresentare 2^{64} valori
  - in matematica, in OGNI intervallo, comunque piccolo, ci sono INFINITI reali
  - se dividiamo l'intervallo dei valori rappresentati con 8 bytes in 2^{64} intervallini, TUTTI i valori matematicamente presenti in ciascun intervallino vanno persi!!
  - questi valori vanno approssimati con il reale più vicino
  - maggiore è il numero di bits usati, minore è l'errore (perché l'intervallino è più piccolo), che comunque, non si può eliminare

Un caso di studio: manipolazione dei reali in C (3)

Esempio:

consideriamo il seguente programma:

```
#include <stdio.h>
int main()
{
  float x;
  x=1.0/10.0;
  printf("x=%f", x);
}
```

Output: x=0.100000

non è un errore: dipende dalla rappresentazione dei reali!

Variante:

```
#include <stdio.h>
int main()
{
  float x;
  x=1.0/10.0;
  printf("x=%.12f", x);
}
```

indica la precisione (di default è 6)

Output: x=0.100000001490

Un caso di studio: manipolazione dei reali in C (4)

Attenzione:

le operazioni sui reali possono portare sorprese:

```
#include <stdio.h>
int main()
{
  float x,somma=0.0; int i;
  x=1.0/10.0;
  for (i=1; i<=1000; i++)
     somm+=x;
printf("somma=%f", x);
}
```

l'errore, che prima compariva solo con alta precisione, ora si è amplificato, e compare anche in precisione standard!

Output: somma=99.999046

Un caso di studio: manipolazione dei reali in C (5)

In sintesi:

- la rappresentazione dei reali porta inevitabilmente a delle approssimazioni, che possono bilanciarsi oppure cumularsi in modo imprevedibile
  - questo non è un errore, ma un fatto intrinseco della rappresentazione con un numero finito di bits
- Come risolvere:
  - tener conto del problema quando si scrive un programma, ad es. invece di testare se x=y, dove sia x che y sono reali, meglio testare se |x-y|<Epsilon, con Epsilon sufficientemente piccolo
  - usare le librerie a precisione "infinita" BigDigits, che in realtà allocano dinamicamente lo spazio per i reali usati, ma attenzione: rallentano l'esecuzione!