2D1458, Problemlösning och programmering under press

Föreläsning 2: Avlusning och antilustekniker

Datum: 2007-09-11
Skribent(er): Emil Hesslow, Stefan Pettersson
Föreläsare: Per Austrin

Föreläsningen handlade om hur man på bästa sätt finner och åtgärdar buggar.

Även om de flesta av tipsen är generella så är de framförallt beskrivna med avseende på problemuppgifter i kursen.

I de fall något är systemspecifikt utgår dokumentet från att du använder skolans UNIX datorer. Med lite efterforskning går dock motsvarande med största säkerhet att även utföra under Windows.

1 Motivering

För att vara en bra programmerare måste man vara bra på att hitta och åtgärda fel. I kursen får man väldigt lite information när något går fel i ens program. Det kan kännas orättvist, men det är inte orimligt.

I många kritiska applikationer är det viktigt att när väl programmet är driftsatt så får inga fel inträffa. Inträffar fel kan det få ödesdigra konsekvenser, dessutom kanske man inte får någon information om felet. Extremfallet är t.ex. styrsystemet för en rymdraket.

En annan mer naturnära situation är kunder som ger bristfälliga felrapporter. De säger att något inte fungerar men kan inte förklara så mycket närmare.

Är man duktig att hitta fel med lite information om felet blir man automatiskt duktig på att hitta fel med mer information. Det omvända är dock inte sant.

Nedan följer lite tips som kan vara av nytta både innan du skickar in en uppgift för rättning såväl som när du måste felsöka. Ytterligare beskrivs verktyg för debuggning och profilering.

2 Innan du skickar in

När du anser dig programmerat klart en uppgift och skall skicka in den till KATTIS (KTH Automated Teaching Tool) bör du gå igenom ett par punkter för att försäkra dig om ett lyckat utfall av rättningen.

2.1 Kompilera med varningar

Kompilera din kod och försäkra dig om att inga problem eller varningar uppstår. Använder du C/C++ kan du använda kompilatorflaggan `-Wall` för att få ytterligare varningar. Åtgärda dessa för att försäkra dig om att de inte resulterar i problem.

2.2 Testa med exempelfallen i uppgiften

Se till att ditt program ger rätt svar och formatering på utskriften för de givna exempelfallen i uppgiften.

2.3 Skriv egna exempelfall

Konstruera egna exempel och se till att även de fallen fungerar utan problem.

2.3.1 Minimala fall

Beroende på problem kan det minimala extremfallet t.ex. vara 0, 1, null, tom sträng, tom lista eller graf utan kanter. Se till att inget kraschar och svaret är rätt.

2.3.2 Maximala fall

Testa med värden som ligger nära och på gränsen till vad som är tillåtet, t.ex. -2^{31} och $2^{31} - 1$ för en integer. Pröva också indata som motsvarar värsta fallet för din lösning, t.ex. långa strängar med samma tecken, stora intervall, sortera lista med samma värden med quicksort.

Dessa fall kan vara svåra att veta svaret på vilket gör resultatet svårare att verifiera. Du kan dock se hur lång tid det tar att köra, om programmet kraschar och åtminstone att svaret som ges är rimligt.

2.3.3 Små fall

Fördelen med små fall är att svaren är lättare att verifiera för hand.

2.3.4 Generera stora slumpfall

Skriv ett program som genererar slumpfall. Även om det kan vara svårt att veta rätt svar kan det användas för att se att svaren åtminstone är rimliga och att programmet inte kraschar.

2.3.5 Planterade svar

För vissa uppgifter kan man modifiera ett slumpfall och ”plantera” ett svar, t.ex. en kort stig i en graf där uppgiften är att hitta kortaste vägen. Med ett planterat svar är det lättare att se om svaret man får är rimligt.

2.3.6 Naiva lösningar

Om problemuppgiften kan lösas naivt kan en sådan lösning, om än långsam, användas för att kolla vad svaret ska bli på dina testfall.

3 Att testköra

Detta stycke gäller speciellt för problemuppgifterna i kursen och är således mindre tillämpbart vid t.ex. testning av en GUI applikation.

Kör inte programmet interaktivt, läs istället indata från en fil. Om du vill kan du även dirigera om utskrifter till en egen fil. Exempel:

```
$ program < filin > filut
```

Ett vanligt fel är att man glömmer skriva ut nyradstecken. Kör man programmet interaktivt kan det ibland vara lätt att missa detta, kör man med indata från fil syns det tydligt att något är fel i utskriften. Dessutom är det lätt hänt att man råkar skriva något litet fel när man anger indata för hand. Om man skriver in testdata i en fil slipper man det problemet.

Att skriva utdata till fil kan vara behändigt när det är mycket som skrivs. Det kan också användas för att se om utdata förändrats mellan två körningar då du gjort en ändring i koden. För att enkelt se skillnaderna mellan två filer kan man använda UNIX-kommandot `diff`.

4 Efter att det blivit fel

4.1 Läs lydelsen

Försäkra dig om att du verkligen förstår uppgiften och vilken indata som förväntas. Ibland är det lätt att anta för mycket och missa något viktigt som man måste ta hänsyn till.

4.2 Läs dokumentationen

Är du osäker på ett funktionsanrop i något bibliotek, se till att läsa dokumentationen noga så du vet att funktionen gör vad du förväntar dig.

4.3 Använd invarianter (assert)

Använd assert i din kod för att bekräfta att ett villkor är sant. Ifall villkoret evalueras till falskt kommer programmet att avbrytas med ett felmeddelande.

Anta inte att variabler har vissa värden. Är du minsta osäker använda istället asserts. Använd med fördel asserts i början av en funktion för att verifiera att alla inkommande variabler är som förväntat, samt i slutet av funktionen för det som returneras.

Typiska användningsområden kan vara att försäkra sig om att ett index är inom rätt intervall, att strängar inte blir längre än förväntat, m.m.

En god vana är att införa asserts medan du programmerar. Då är det lättare hålla reda på vilka variabelvärden som förväntas än att göra det långt senare när du glömt hälften.

```
C/C++:
  assert(villkor);


Java:
  assert (villkor) : "felmeddelande";
```

4.4 Spårutskrifter

Spårutskrifter används för att vid en given position i programmet skriva ut värden på kritiska variabler. Lägg till utskrifter på väl valda ställen och se till att det tydligt framgår av utskriften var någonstans i koden de görs.

Gör utskrifterna till `STDERR` för att lättare se dem. Saker som skrivs till `STDERR` ignoreras av KATTIS, men:

- Det som skrivs ut hamnar i KATTIS utskriftsbuffert som är begränsad, skrivs för mycket ut kan du få felmeddelandet `OUTPUT_LIMIT_EXCEEDED`.
- Programmet kan exekveras långsammare om det är mycket som skrivs ut.

För att hantera spårutskrifter i C/C++ är det praktiskt att använda ett eget makro.

4.4.1 C++

```
#define dout debug && std::cerr

bool debug = false;

// Exempel
dout << "LOOP: " << i << ", " << j << std::endl;
```

Om man inte vill ha debugutskrifter för hela programmet kan man sätta debug till true först när det är intressant, t.ex. när en variabel får ett visst värde.

4.4.2 C

```
#include <stdio.h>
#define dprintf debug && fprintf
int debug = 0;

// Exempel
dprintf(stderr, "loop: %d, %d\n", i, j);
```

4.4.3 Java

```
boolean debug = false;

void dprintln(String s) {
  if (debug)
    System.err.println(s);
}

// Exempel
dprintln("LOOP: "+ i +", "+ j);
```

5 När vi inte har en aning om vad som går fel

5.1 Kraschar på KATTIS

Försök identifiera var.

Prova med att kommentera bort olika delar av koden. Ett första steg kan vara att kommentera bort hela lösningen och endast ha kvar inläsning och utskrift för att isolera om problemet finns där.

Tänk på att ett fel oftast fortplantar sig, så även om det tar sig uttryck på ett visst ställe kan det vara tidigare i koden något händer som resulterar i felet.

Tänk också på beroenden mellan olika delar av koden, kommentera inte bort en del av koden som senare delar av koden är beroende av.

5.2 Kompilatorvarningar

Kompilera med `-Wall` (C/C++) för att se kompilatorvarningar. Åtgärda dessa.

5.3 Optimeringsflaggan

Testa med optimering både avslagen och påslagen vid kompilering.

Det är möjligt att ett fel bara inträffar i ett visst läge alternativt att symptomen endast visar sig i ett läge. Det kan vara svårt att använda en debugger med optimering påslagen.

5.4 Läs igenom koden och snygga till

- Skriv om koden till mer lättläst
- Strukturera allt så det blir tydligt vad som görs
- Ta bort oanvänd kod

I många fall kan en bugg försvinna tack vare detta, ibland utan att man vet riktigt varför.

Det är ofta farligt när en bugg försvinner utan att man vet varför, ofta har buggen kanske inte försvunnit utan bara symptomen. Man bör därför alltid försöka att förstå vad som egentligen gick fel.

5.5 Förklara koden för någon annan

Att förklara för någon annan vad koden gör och varför den fungerar leder ofta till att man själva kommer på varför den inte fungerar.

Har du ingen annan att förklara för, berätta för en nallebjörn eller dylikt. Det fungerar lika bra bara du gör det på allvar.

5.6 Ta en paus

Sitter man för länge är det lätt att man stirrar sig blind på koden och inte ser eventuella problem. Gå iväg och gör något annat (eller byt uppgift) för att åter titta på koden med utvilade ögon och klar hjärna.

5.7 Gör om - gör rätt

Ibland kan det vara lika bra att helt enkelt börja om på nytt från början. Om det tog 30 minuter att skriva koden men du spenderat 2 timmar på att hitta fel, då kan det vara väl spenderad tid att skriva om allting från början.

5.8 Statisk analys

Testa ett program som gör en statisk analys av din kod. För Java finns t.ex. FindBugs: `http://findbugs.sourceforge.net`

6 Diverse vanliga buggar och tips

6.1 Initiering av variabler

Se till att initiera alla variabler, speciellt arrayer. Symptom kan vara att samma test som körs två gånger i rad ger olika svar.

6.2 För små arrayer

I C/C++ är strängar nullterminerade vilket tar en plats. Se till att dina strängar inte är för korta.

- Tumregel: addera lite extra utrymme för säkerhets skull

6.3 För små datatyper

- Tumregel: Är du nära gränsen så välj en större datatyp

På KATTIS	bitar
short	16
int	32
long (C/C++)	32
long (Java)	64
long long (C/C++)	64

6.4 Flyttal

- Tumregel 1: Undvik om du kan (använd heltal om det går)
- Tumregel 2: Om du använder, använd double (ej float) för högre precision

6.5 Numeriska konstanter

När du använder numeriska konstanter i C/C++ bör du använda lämpligt suffix för att försäkra dig om att de tolkas som den datatyp du önskar. Om inte kan det lätt inträffa oönskade avrundningsfel och liknande precisionsbortfall.

integer	42
double	42.0
unsigned int	42U
long	42L
unsigned long	42UL
long double	42.14159L
float	42.02e23f

6.6 Trådsynkronisering

Använder du trådar måste du se till att de är synkroniserade där det krävs (ej aktuellt på denna kurs).

6.7 Alignmentfel

En integer måste vara lokaliserad på en adress som är en multipel av 4 (beroende på plattform). Fel resulterar i bus error.

6.8 Editor

Använd en bra editor med stöd för syntax highlighting och liknande stödfunktioner. Det gör att det är lättare att se saker som oavslutade parenteser och liknande slarvfel.

6.9 Sök på felmeddelanden

Får du ett felmeddelande du inte förstår dig på är sannolikheten att du inte är den första som undrar. Använd valfri sökmotor och sök på felmeddelandet så är chansen stor att du finner något som kan hjälpa dig.

7 Debugger

Ibland kan det vara användbart att använda en debugger för att hitta var någonstans det går fel. T.ex. om programmet kraschar kan man via debuggern få reda på var exakt kraschen sker och vilka värden olika variabler hade när det inträffade och på så sätt kunna lista ut varför det går fel. En debugger man kan använda är DDD som egentligen bara är ett grafisk gränssnitt till GDB (C/C++) och JDB (Java).

Vill man debugga ett C/C++ program måste man först kompilera programmet med `-g` flaggan. Detta för att debuginformation ska läggas in i programmet.

```
$ g++ -g -o [filename] [filename].cpp
$ ddd [filename]
```

Eftersom `-g` gör att programmet kompileras annorlunda kan det i värsta fall vara så att en krasch som inträffar normalt inte inträffar med `-g` flaggan.

I de flesta debuggers har man bland annat möjlighet att steppa igenom programmet, använda breakpoints, se värden på variabler samt använda backtrace för att se varifrån funktionen anropades.

Step gör det möjligt att exekvera en rad kod eller en instruktion åt gången. På så vis kan man stega igenom programmet och se vad som händer en rad i taget.

Med breakpoints är det möjligt att stanna exekveringen av programmet vid ett givet ställe. Det är användbart om man vill exekvera i full hastighet till ett visst ställe och stanna först där.

Backtrace ger dig information om vilka funktionsanrop som gjorts för att nå den aktuella platsen i koden. Detta är användbart då en krasch kan ske i en funktion som du själv inte skrivit utan anropats i flera steg t.ex. via en biblioteksfunktion.

8 Profiler

Om exekveringen tar lång tid och man vill veta vad det är som tar lång tid så kan man använda sig av en profiler.

8.1 C/C++

Ett exempel är `gcov` som man kan använda för att se hur många gånger varje rad i koden körts. Detta är användbart av två anledningar. För det första kan man se vilka rader som körs oftast. Att en rad exekverats flest gånger innebär inte automatiskt att det är den som gör programmet långsamt, dock kan det vara en bra indikation på var det kan vara värdefullt att börja optimera.

För det andra ser man även vilka rader som aldrig körts. Har en sektion kod aldrig exekverats kan den mycket väl innehålla en bugg som du själv inte upptäckt. Se till att skriva ett testfall som även använder sig utav den delen av koden.

För att använda `gcov` börja med att kompilera det med flaggorna `-fprofile-arcs` och `-ftest-coverage`.

```
$ g++ -fprofile-arcs -ftest-coverage -o [filename] [filename].cpp
```

Kör sen programmet som vanligt.

```
$ ./[filename] < input
```

När programmet körs kommer filerna `[filename].gcda` och `translate.gcno` att skapas. Kör sen `gcov` för att skapa `[filename].cpp.gcov`.

```
$ gcov [filename]
```

`[filename].cpp.gcov` är en textfil som kan se ut som följer:

```
...
function main called 1 returned 100% blocks executed 95%
        1:    5:int main(void) {
        1:    6:        char message[10010];
...
```

Första siffran på varje rad talar om hur många gånger den raden kördes. I det här fallet kördes båda raderna en gång var. Om raden inte körts alls kommer det indikeras med `#####` istället.

För att ta reda på hur stor del av körtiden varje funktion tog kan man använda sig av `gprof`.

8.2 Java

Ett profiler-alternativ för Java är JRat: `http://jrat.sourceforge.net`