2D1458, Problemlösning och programmering under press

Föreläsning 2: Debugging

Datum: 2006-09-11
Skribent(er): Marcus Dicander, Alix Warnke
Föreläsare: Fredrik Niemelä

Föreläsningen handlade om felsökning och hur man ska förebygga fel när man programmerar. Vi studerade tre olika typer av fel: Kompileringsfel, Kraschfel och Logiska fel.

1 Återskapa felet med hjälp av bra testfall

Om ditt program kraschar av någon oförklarlig anledning så behöver du testfall för att återskapa felet och därefter hitta och eliminera det. För att provocera fram en bugg behöver man en bra uppsättning testfall som om möjligt ska innehålla exempel från var och en av kommande kategorier.

1.1 Triviala fall

För att kontrollera att programmet klarar basfallen. Klassiska basfall är 0, 1, tomma strängen, tomma listan och null.

1.2 Små instanser

Små instanser är värdefulla eftersom man kan verifiera för hand att svaret är korrekt.

1.3 Stora instanser

Med stora instanser kan det vara svårt att avgöra om svaret är helt korrekt, men det bör gå att avgöra om svaret är rimligt. Dessutom kan man passa på att kontrollera om programmet uppfyller tidskravet.

1.4 Extremvärden

För att framkalla krasch eller oväntat beteende. Typiska extremvärden är maxint och minint. Ibland är det bara extremvärden som får programmet att krascha eller göra fel.

1.5 Corner case

Ett corner case uppkommer när flera variabler antar sina extremvärden samtidigt. Precis som vid extremvärden så kan det vara så att programmet bara gör fel just här.

1.6 Oväntad indata

En del program visar bara sina sårbarheter vid oväntade fall. Som exempel kan en dålig quicksort-implementation krascha när den sorterar en lista med tal som alla antar samma värde.

1.7 Hittar inte felet ändå?

Om man trots dessa brutala men nödvändiga testfall fortfarande inte kan hitta felet så kan det bero på något av följande:

- Glömt att synkronisera trådar (dock ovanligt i denna kurs eller på programmeringstävlingar)
- Glömt att initialisera variabler (kan orsaka oväntat beteende)
- Alignment-problem: Minne kan behöva allokeras på en adress som är en jämn multipel av 2 för att sedan läsas, detta problem är inte särskilt vanligt

2 Lokalisera felet utan debugger

Det gäller att här skilja på kompileringsfel (enklast att hitta och åtgärda), körningsfel (krascher), och logiska fel (Datorn gör exakt vad du sagt åt den att göra, men inte vad du vill att den ska göra). Vid kompileringsfel skriver kompilatorn ut på vilken rad felet ligger. Tänk på att läsa kompilatorns felmeddelanden uppifrån eftersom en del fel fortplantar sig nedåt. Vid körningsfel skriver Java ut vilken rad som orsakat kraschen. C och C++ gör det inte men vi kommer att titta på tekniker för att snabbt och definitivt ta reda på var i koden som programmet kraschade.

2.1 Binärsökning...

Binärsökning kan användas till att lokalisera fel genom att upprepade gånger halvera sökintervallet.

2.1.1 ...med bortkommentering av kod

Binärsök genom koden genom att kommentera bort stora stycken. Kommentera bort hälften av koden och se om det fortfarande kraschar. I så fall, upprepa proceduren om och om igen.

2.1.2 ...med spårutskrifter

Lägg in utskrifter på väl valda platser i koden (mitt i sökintervallet är väl valt) som gör det lätta att avgöra (1) var i koden det skrevs ut och (2) värden på kritiska variabler. Använd preprocessorn för att hantera spårutskrifter i C. Skriv detta längst upp i implementationsfilen.

```
#ifdef _DEBUG
#define DEBUG(X) X
#else
#define DEBUG(X)
#endif
```

Skriv spårutskrifter så här:

```
DEBUG(printf("LOOP: i = %d, j = %d\n", i , j));
```

Se till att flusha spårutskrifter när du jagar kraschbuggar. Om du inte gör det så kan programmet ha passerat en utskriftsrad trots att du inte ser den på skärmen. I C skriver du:

```
fflush(stdout);
```

Använd preprocessorn för att hantera spårutskrifter i C++. Skriv detta längst upp i implementationsfilen:

```
#ifdef _DEBUG
#define dout debug && std::cerr
#else
#define dout if(false) std::cerr
#endif
```

Skriv spårutskrifter så här:

```
dout << "LOOP: i = " << i << " j = " << j << std::endl;
```

Om _DEBUG inte är definierat kommer ovanstående rad att ersättas med ett if(false) i början. Eftersom if(false) aldrig kan vara sant så kommer raden inte att skrivas ut. För att flusha buffern skriver du:

```
std::cerr.flush();
```

och i Java skriver du långt uppe i klassen (ändra till true vid felsökning):

```
private final int debug=false;
```

Spårutskrifter på formen:

```
if(debug){
    System.err.println("i = " + i + " j = " + j);
}
```

Flusha genom att:

```
System.err.flush();
```

2.2 Assert

Använd *assert* för att kontrollera antaganden om variabelvärden och minnesåtgång. Är strängen kortare än en Shakespearepjäs? Är listan längre än ett schackparti mellan Kasparov och Carola? Är din signed int fortfarande inte negativ?
I C/C++ skriver du:

```
assert(villkoret)
```

Om *villkoret* inte är uppfyllt så stannar exekveringen på den raden.
I Java skriver du:

```
assert(villkoret)[:"felmeddelande"];
```

Om *villkoret* evalueras till false så skickas *felmeddelande* till ExceptionHandler.

3 Lokalisera felet med debugger

På skolan finns många debuggers tillgängliga, bland annat: gdb (för C/C++), ddd (grafiskt skal för gdb) och jdb (för Java). Nedan beskriver vi gdb som ett exempel på en debugger. Andra debuggers har likande kommandon.

För att kunna använda gdb måste ditt program kompileras med flaggan -g.

```
$ gcc -g <filename>.cpp
$ gdb a.out
```

Här är en lista på kommandon i gdb:

Kommando	Förtydligande
r (run)	
l (list)	l <radnummer>, l<filnamn>:<rad>
b (breakpoint)	b<rad>, b<funktionsnamn>
cont (continue)	kör koden tills den stöter på en breakpoint eller slutet av programmet
d/p (display/print)	d <x>, p <x> skriver ut värdet av x
und (undisplay)	und <x> slutar visa variablen x
step	step utför en rad instruktioner - om ett funktionsanrop, en rad inuti den
n (next)	next utför en rad instruktioner - om ett funktionsanrop, kör hela och returnerar
bt (backtrace)	Backar körningen till förra breakpoint
del (delete)	del N, där N är en breakpoint, endast del tar bort alla breakpoints
i (info)	info stack ger t.ex information om stacken
set	set variable x=12, tilldelning
cal (call)	call foo() anropar funktionen foo()
def (define)	define bar
h (help)	hjälp om andra kommandon
fin (finish)	avslutar gdb

Ofta vill man placera ut breakpoints i början av funktioner så att exekveringen stoppas när man nått så långt. Vill vi stanna före funktionen Foo() så skriver vi: *br Foo*. gdb kommer då ge feedback om var denna brytpunkt sattes in, exempelvis: *Breakpoint 2 at 0x2290: file main.cpp, line 30*
Det går även alldeles utmärkt att istället för ett funktionsnamn ange en rad i koden, ex: *br 20* Prova nu att köra kommandot run, exekveringen kommer att stanna vid varje breakpoint. Vill du ta bort en breakpoint skriver du helt enkelt *delete N*, där *N* är id-numret på breakpointen som ska bort. Har du väldigt många breakpoints då ett id-nummer inte säger dig särskilt mycket kan du alltid skriva: *info break* för att få detaljer. En bra nybörjartutorial finner du på t.ex:
`http://www.cs.princeton.edu/~benjasik/gdb/gdbtut.html`

4 Ytterligare tips för att motverka fel

Naturligtvis vore det bästa att inte skriva fel från början, så vi börjar där.

4.1 Förebyggande

För att spara tid och slippa onödig felsökning under programmeringsfasen så kan du förbereda dig på flera sätt.

4.1.1 Design by contract

Design by contract är en programutvecklingsmetod där preconditions och postconditions kan ses som ett kontrakt som en metod eller funktion måste uppfylla. De variabler och datastrukturer som en metod eller funktion behandlar antas ha vissa egenskaper. Dessa egenskaper kan man ange explicit i form av assertions. Om någon av dessa antaganden inte har uppfyllts så har vi ett kontraktbrott och programexekveringen avbryts omedelbart och obönhörligen.

4.1.2 Använd en bra editor

En bra editor brukar ha tydlig syntax highlighting och stöd för kompilering och exekvering, men framför allt är en bra editor någon som du är van att arbeta med och som du känner till väl.

4.1.3 Vanliga fel

Lär dig vilka fel du brukar göra. Sök med Google efter vanliga fel i just ditt programmeringsspråk.

4.1.4 Post Mortem

Se till att förstå de problem som uppkom och hur du löste dem. Det är förebyggande inför nästa gång.

4.2 Under felsökningen

Buggar är illmariga och kan ha överlevt dina förebyggande insatser, men oroa dig inte - Vi har tips för även denna fas!

4.2.1 Läs dokumentationen

Bra dokumentation är skriven av kloka människor som av erfarenhet eller insikt lyckats förutspå många av de problem som kan uppstå. Det kan vara en idé att läsa den ordentligt för felet kan bero på att man missförstått en dokumenterad detalj.

4.2.2 Läs och förstå problemet samt indata och utdata

Ett välskrivet problem beskriver i detalj vilka antaganden man kan och inte kan göra om indata. De enklaste buggarna bygger på att man läser in eller skriver ut på fel sätt. Lite knepigare är om indata kan anta mycket stora värden. Mycket knepigare är om man har en algoritm som löser problemet men den är för generell

och långsam, så programmet spräcker tidsgränsen. Eller åt andra hållet: Man kan ha gjort för många antaganden om indata så att algoritmen bara löser vissa specialfall av problemet.

4.2.3 Dialogmetoden

Sokrates förespråkade att använda dialoger för att lära ut abstrakta koncept. Tag lärdom av detta och försök förklara ditt program eller din idé noggrannt för någon annan. Det kan öka bådas förståelse för problemet. Många självklara fel kommer fram när man förklarar programmet för någon annan. Under förklaringen så kanske du själv upptäcker saker du inte riktigt har förstått eller antaganden du gjort som inte stämmer och då vet man var man ska rikta sin energi. Det är också viktigt att få bra feedback på sina förklaringar. Om man lär sig vilken typ av frågor som brukar ställas eller vad frågeställaren brukar påpeka så lär man sig förklara på ett bra sätt från början.

4.2.4 Papper och penna

Papper och penna är enkla, klassiska, beprövade och mycket effektiva verktyg för felsökning. Skriv ut koden. Många har lättare att förstå den från ett papper än från skärmen. Under tävlingspass innebär det också att datorn blir ledig för att lösa andra problem medans du jagar buggar. Stega igenom och räkna i huvudet. Gör bra anteckningar för att avlasta närminnet. Skriv misstänkta algoritmer i pseudokod för att kolla om de verkligen fungerar.

4.2.5 Djupare fel

Fastna inte i att ändra småsaker. Ibland kan programmet ha ett djupare problem (fel algoritm, fel idé). Spara undan den gamla filen under ett annat namn och testa att göra större ändringar. Om det senare visar sig att den gamla lösningen var bättre kan man enkelt gå tillbaka.

4.2.6 Ta pauser

Ibland blir man så låst att man inte kommer någonstans. Då kan det vara bra för såväl hälsa som problemlösningsförmåga att ta en paus och återvända till problemet senare. Hjärnan kan bearbeta problem trots att man inte anstränger sig för att tänka på dem.

4.2.7 Ändra bara en sak i taget

Om du utgår från ett program som du redan förstår någorlunda och sedan gör många ändringar innan du kompilerar om och testar det så kan det bli svårt att lokalisera nya fel som du råkade introducera. När man ändrat 5 saker som verkade helt galna så kanske programmet plötsligt gör fel på testfall som det tidigare klarade. När man letar svårfunna fel under tidspress är det sista man vill att introdcuera ännu fler svårfunna fel.