

Debugging

Introduzione

Talvolta i programmi che scriviamo non funzionano, oppure forniscono risultati che non ci aspettiamo

Un errore in un programma può non pregiudicarne la compilazione ma impedirne il corretto funzionamento

La causa di questi malfunzionamenti sono spesso errori difficili da individuare, come ad esempio un `=` al posto di `==`

Più è grande il programma, più è difficile trovare gli errori

Debugging

Il **debugging** di un programma è necessario quando *il software non si comporta come ci aspetteremmo*, vale a dire quando i risultati sono differenti da quelli desunti dalle specifiche

In genere ciò non accade perché il software è pieno di errori, ma solo a causa di un piccolo frammento di codice scritto in maniera non corretta

bug: frammento di codice che non soddisfa le specifiche

debugging: ricerca del frammento di codice che non soddisfa le specifiche

Debugging



*"LISTEN, KID, HOW MANY
TIMES DO I HAVE TO
EXPLAIN IT TO YOU...BUGS
DON'T BECOME
PROGRAMMERS."*

Debugging

Cosa accade se nel **software** c'è un **bug**?

- **errore** di sintassi/semantica dal **compilatore** (?)
- **interruzione inattesa** del programma (forse...)
- il programma **non si ferma** più
- il programma termina dando **risultati sbagliati**

```
void main(int argc, char *argv[])
{
    int sum, int i;
    for (i = 1; i <= argc; i++)
        sum += Integer.parseInt(argv[i]);
    System.out.println(sum);
}
```

Debugging

Il debugging di un programma viene eseguito in **tre fasi**:

- **trovare le istruzioni** che causano l'errore
- **capire** il perché
- **eliminare** il problema

Capire l'errore è relativamente semplice, una volta trovato

Eliminare il problema è molto semplice quando l'errore è stato trovato e capito

Debugging

Isolato il frammento di codice, per capire un errore bisogna:

- analizzare lo stato prima dell'esecuzione
- analizzare lo stato dopo l'esecuzione
- confrontare questi stati con quello dettato dalle specifiche
- seguire l'esecuzione e trovare dove il cambio di stato non è quello previsto

Per stato di un programma intendiamo l'insieme dei valori di tutte le variabili attive

Debugging

La vera difficoltà sta nel capire *dove* si trova l'errore

Poiché l'errore è un frammento di codice che fa qualcosa di inaspettato, è necessario

- capire in dettaglio cosa dovrebbe fare
- capire in dettaglio cosa fa

E' necessaria una strategia che permetta di focalizzare l'errore e i dati connessi con esso

Strategia di ricerca

Durante l'esecuzione il programma incontra l'istruzione sbagliata e i risultati diventano errati (*non è detto che il programma si interrompa bruscamente in questo momento*)

Bisogna identificare il blocco di istruzioni che altera lo stato in maniera inattesa

E' necessario conoscere lo stato in vari momenti durante l'esecuzione

Strategia di ricerca

Una semplice strategia per scoprire l'errore può essere:

- **inserire** un'istruzione di **stampa** dei valori delle variabili nel mezzo della sezione sospetta
- se i **valori non** sono **corretti**, l'errore è nella **prima metà**, nella seconda metà altrimenti
- si continui applicando l'algoritmo di bisezione

Strategia di ricerca

Ad ogni passo viene eliminato metà programma

In un numero sufficiente di passi verrà individuata l'istruzione errata

Problemi:

- quali variabili stampare, tutte? E se sono molte?
- qual è il mezzo della sezione sospetta?
- è sicuro che in questa maniera troviamo l'errore?

Strategia di ricerca

Un'altra strategia potrebbe essere di **visualizzare** le strutture dati:

- appena inicializzate
- in *punti strategici*
- alla fine dell'esecuzione

I *punti strategici* possono essere, ad esempio, dopo:

- il primo, il secondo e l'ennesimo ciclo del main
- il "premi un tasto" che fa terminare improvvisamene il programma
- l'ultimo punto dove il programma riesce a stampare

Esame dello Stato del Programma

Uno **strumento utile** è un meccanismo che permetta di **visualizzare lo stato** di un programma

Chiedendo al programma di **stampare le variabili sospette**:

- si **altera il programma**
- bisogna sapere **quali** sono le **variabili sospette**
- si rischia di stampare **troppo** (e di avere bisogno di un altro programma per analizzare i dati!)

Esame dello Stato del Programma

Un'alternativa sarebbe uno **strumento** che permetta:

- di **accedere allo stato** di un programma, interrompendo l'esecuzione dove si vuole
- di **ispezionare parti dello stato**

Un programma sì fatto si chiama **debugger** e permette di analizzare lo stato di un programma

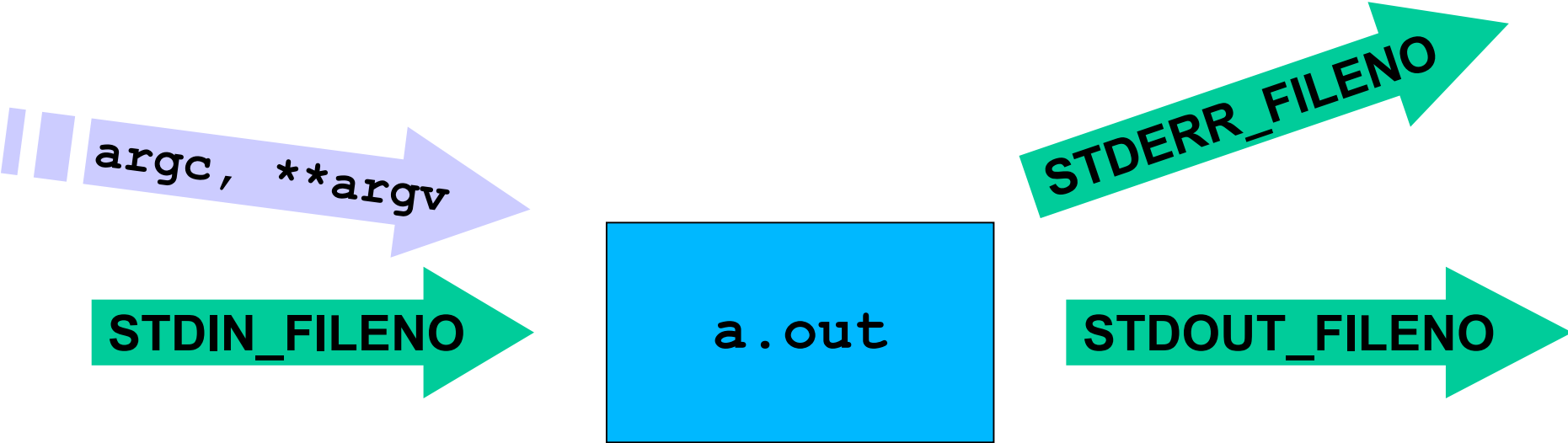
Esecuzione di un programma C

In ambiente Unix, un programma C è in esecuzione fino a che:

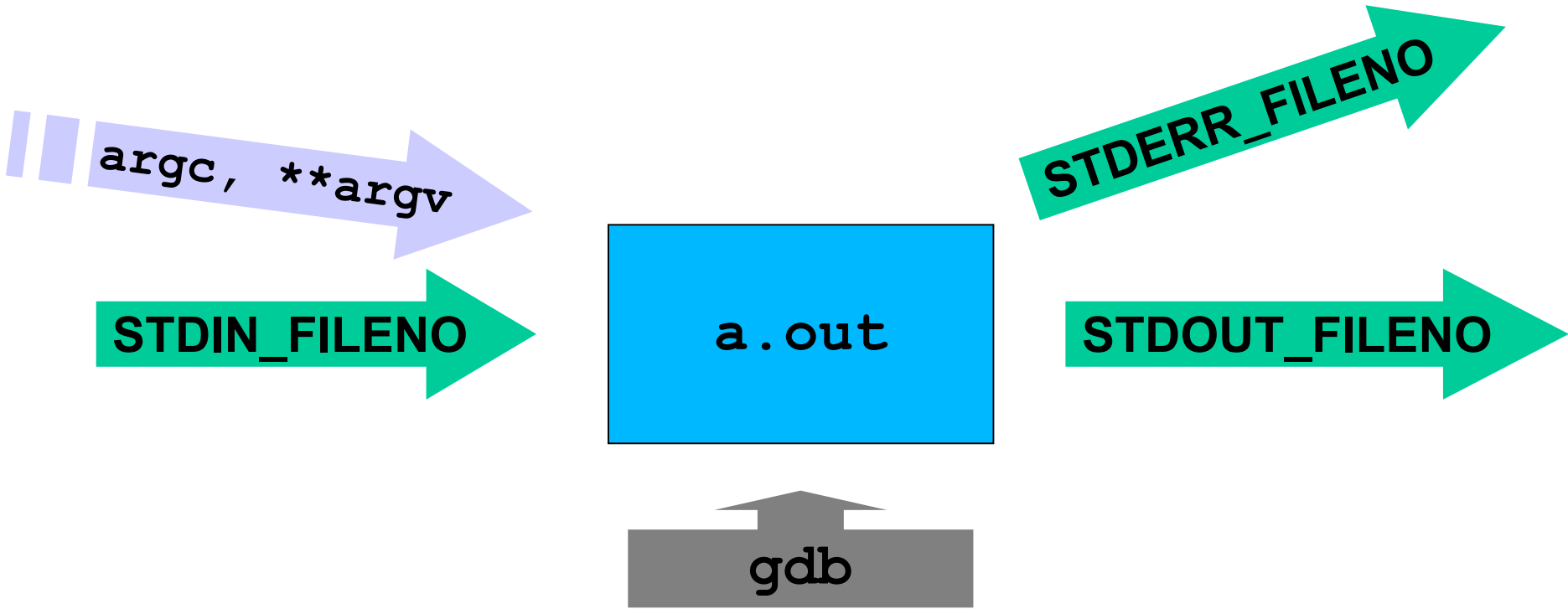
- **finisce**, producendo i suoi risultati
- si accorge di un errore e **chiama `exit()`**
- **riceve un segnale** per un errore (es. segmentation violation)

Anche i **programmi interpretati** (Java, Perl, script di shell,...) hanno un **comportamento analogo**

Esecuzione di un programma C



Esecuzione con debug



Debugger

Un **debugger** permette di eseguire:

- un' **esecuzione** normale (**run, cont**)
- uno **stop** in un determinato punto (**break**)
- un' **istruzione** alla volta (**step, next**)
- **esaminare** lo stato di un programma (**print**)

Il debugger gdb

gdb è un debugger a linea di comando per programmi C & C++

Esistono diverse interfacce grafiche (xgdb, ddd, ...)

Se si utilizza un IDE, il debugger è di solito fornito tra gli strumenti

Tutti i debugger forniscono strumenti di esecuzione controllata e visualizzazione dello stato

Alcuni permettono di utilizzare strumenti grafici per la visualizzazione delle strutture dati

Il debugger **gdb**

Per utilizzare il debugger **gdb** è necessario compilare i programmi con l'opzione **-g**

gdb ha due argomenti sulla linea di comando:

```
$ gdb -c core eseguibile
```

L'argomento **core** è opzionale

gdb: comandi di base

quit	esci da gdb
help [CMD]	help on-line
CMD	fornisce informazioni sul comando
run ARGS	esegue il programma

\$ xyz < data

è ottenuto con:

(gdb) run < data

gdb: comandi di stato

where

Trova quale funzione stava eseguendo il programma quando si è interrotto

list [LINE]

Mostra cinque linee da ciascun lato dell'istruzione corrente

print EXPR

Mostra il valore corrente delle variabili (**a@1** mostra tutto l'array **a**).

gdb: comandi di esecuzione

break [PROC|LINE]

All'ingresso nella procedura PROC (o alla linea LINE), interrompi l'esecuzione e ritorna il controllo a gdb

next

Segui la prossima istruzione; se l'istruzione è una chiamata a procedura, esegui l'intero corpo della procedura

step

Esegui la prossima istruzione; se l'istruzione è una chiamata a procedura, vai alla prima istruzione nel corpo della funzione

Uso del debugger

Una volta individuata la regione in cui si trova l'errore:

- si inserisce un breakpoint subito prima,
- si esegue il programma di nuovo con gli stessi dati
- si va avanti un'istruzione alla volta nella sezione critica
- si controllano i valori delle variabili sospette ad ogni passo

Questi passi permetteranno probabilmente di individuare una variabile con un valore errato

Uso del debugger

Una volta trovato che il **valore** di una determinata variabile (es. **x**) è **errato**, bisogna capire il perché

Le possibilità sono due:

- l'**istruzione** che ha assegnato **x** è sbagliata
- altre **variabili** nell'istruzione sono **sbagliate**

Esempio

```
if (c > 0) { x = a+b; }
```

Se **x** dopo questa istruzione è sbagliato e l'espressione è corretta, allora **a** e/o **b** sono state inizializzate con valori errati

Uso del debugger

Eseguire il **debugging** di un'applicazione è un esempio di **applicazione del metodo scientifico**:

- si **enuncia** una **tesi**
- si collezionano **dati** per **verificare** tale tesi
- sulla base dei dati raccolti si **modifica** la **tesi**

Nel caso del debugging la tesi è

"L'errore si trova in quest'istruzione"

Esempio: test.c

```
#include <stdio.h>
int
main()    /* divisione intera tra due numeri */
{
    int    x,y,z;

    scanf("%d %d", &x, &y);
    if ( x = 0 )
        printf("Can't divide by 0!\n");
    else
        z = div ( y, x );

    printf("%d",z);
    exit (0);
}

int div (int y,int x) { return(y/x); }
```

Esempio: compilazione

```
[mariog]# gcc -g -o test test.c
```

```
[mariog]# test
```

```
3 4
```

```
Floating point exception (core dumped)
```

*Non ci sono errori o warning
nella compilazione, ma in
esecuzione
il programma termina
improvvisamente*

Esempio: debugging

```
[mariog]# gdb test  
GNU gdb 5.0
```

```
...
```

```
This GDB was configured as "i386-redhat-  
linux"...
```

```
(gdb)
```

```
(gdb) break main
```

```
Breakpoint 1 at 0x80484b2: file gdb.c, line 5.
```

Procediamo con l'esecuzione interattiva del programma attivando il debugger

Fissiamo un breakpoint all'inizio del programma con il comando

Un **breakpoint** è un punto in cui il **programma** deve essere **fermato** in attesa di ulteriori direttive da parte dell'utente

Esempio: debugging

```
(gdb) run
Starting program:
/home/user/test/test
Breakpoint 1, main () at gdb.c:5
5     scanf("%d %d",&x,&y);
```

```
(gdb) next
8 9
7     if (x = 0)
(gdb)
```

*Per avviare il programma si utilizza il comando **run***

Si arresta al primo breakpoint e visualizza la prossima istruzione da eseguire

*Per eseguirla utilizziamo **next***

Esempio: debugging

```
7      if (x = 0)
(gdb)
10     z = div (y,x);
(gdb)

10     z = div (y,x);
(gdb) step
div (y=8, x=0) at gdb.c:17
17     return(y/x);
(gdb)
```

*Per entrare nella funzione **div** ed eseguirla interattivamente usiamo il comando **step***

*Dai valori notiamo che **x** all'ingresso di **div** è diventata zero*

Esempio: debugging

Con il comando **cont** proseguiamo fino alla fine

```
(gdb)cont
```

```
Program received signal SIGFPE, Arithmetic exception.
```

```
0x8048515 in div (y=8, x=0) at gdb.c:17
```

```
17     return(y/x);
```

```
(gdb) run
```

```
...
```

```
Breakpoint 1, main () at gdb.c:5
```

```
5     scanf("%d %d",&x,&y);
```

```
(gdb) next
```

```
8 9
```

```
7     if (x = 0)
```

```
(gdb)
```

Ma dove e' cambiato il valore di **x**?

Esempio: debugging

```
7      if (x = 0)
(gdb) print x
$1 = 7
```

*Verifichiamo il valore di **x** con il comando **print***

```
$1 = 7
(gdb) display x
1: x = 7
(gdb)
```

***display** ci consente di visualizzare il valore di una variabile ad ogni passo dell'esecuzione*

Esempio: debugging

Nell'esecuzione della riga

```
7      if (x = 0)
```

si verifica il cambiamento di valore

```
7      if (x = 0)
```

```
(gdb) next
```

```
1: x = 0
```

```
(gdb)
```

Questo ci **consente** di rilevare e **correggere** l'errore

File core

Quando l'**esecuzione** di un file viene **arrestata** bruscamente per la presenza di un bug, o per qualsiasi altro motivo, il sistema genera un file **core**

Questo file **contiene** informazioni sullo **stato** nel momento in cui il programma è terminato

File core

```
[mariog]# gdb -c core test
GNU gdb 5.0
...
Core was generated by `test'.
Program terminated with signal 8, Arithmetic exception.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x8048515 in div (y=8, x=0) at test.c:17
17      return(y/x);
(gdb)
```

La lettura del file **core** generato da **test** evidenzia il punto in cui il programma è stato interrotto

Help in linea

(gdb) help

List of classes of commands:

aliases -- Aliases of other commands

breakpoints -- Making program stop at certain points

data -- Examining data

files -- Specifying and examining files

internals -- Maintenance commands

obscure -- Obscure features

running -- Running the program

stack -- Examining the stack

status -- Status inquiries

support -- Support facilities

tracepoints -- Tracing of program execution without

stopping the program

user-defined -- User-defined commands

Alcuni consigli

Eseguire il debugging di un'applicazione può diventare frustrante se:

- l'eseguibile non deriva dal sorgente che si sta leggendo
- le funzioni di libreria non ritornano mai lo stato di errore
- non c'è controllo sui parametri di scambio delle funzioni
- le librerie di sistema contengono errori
- "Il problema *non può* trovarsi in questa funzione!"
- "Il problema *deve* trovarsi in questa funzione"

Massime

Before you can fix it, you must be able to break it

(consistently).

(non-reproducible bugs ... Heisenbugs ... are extremely difficult to deal with)

If you can't find a bug where you're looking, you're looking in the wrong place.

(taking a break and resuming the debugging task later is generally a good idea)

It takes two people to find a subtle bug, but only one of them needs to know the program.

(the second person simply asks questions to challenge the debugger's assumptions)

Zoltan Somogyi,
Melbourne University

Sommario

Il debugger è uno degli strumenti fondamentali per lo sviluppo del software

Esso permette di trovare rapidamente errori che provocano interruzioni improvvise nell'esecuzione del software

Il suo utilizzo permette di verificare il corretto funzionamento delle applicazioni

Link utili

Manuali del gdb:

<http://www.gnu.org/manual/gdb-4.17/gdb.html>

Lucidi sul debugging:

<http://www.cse.unsw.edu.au/~cs2041/lec/testing/slide052.html>

Esercizi

1. Scrivere un programma C che, presi in input uno o più file, verifica se essi sono dei link simbolici e se puntano a dei file esistenti.
2. C'è un numero massimo di sotto/.../sottodirectory che una directory può avere? Scrivere un programma C e commentare i risultati.
3. Scrivere un programma C che, preso come argomento una directory ed una data, modifica la data di ultimo accesso ai dati di tutti i file contenuti nella directory.
4. Inviare a mario.guarracino@unina.it link a risorse interessanti sul gdb