

Il gdb in breve

Autore: Matteo Lucarelli
ultima versione su: www.matteolucarelli.net
Documento in costruzione

Il presente documento vuole essere un breve prontuario degli usi del gdb (GNU debugger). Illustra quindi solo alcune delle innumerevoli possibilità del noto debugger, ed in particolare solo i comandi e le tecniche di uso più comune nella programmazione C e C++.

Per una trattazione completa si veda la documentazione ufficiale GNU.

Come funziona

Il debugger è uno strumento indispensabile per l'analisi di un processo in esecuzione. Permette di eseguire il programma un'istruzione alla volta, di verificare e modificare in tempo reale il valore dei simboli e molto altro.

Perché un binario sia analizzabile tramite debugger deve contenere al suo interno una certa quantità di informazioni. E' quindi necessario che il programma sia compilato con il flag `-g`, o, nel caso specifico di `gcc/gdb -ggdb`:

```
gcc -g ...
```

Disponendo di un binario così compilato sarà quindi sufficiente impartire il comando:

```
gdb [nomebinario]
```

per entrare nel prompt del debugger:

```
(gdb)
```

nel quale sono disponibili tutte le comodità alla quale la bash ci ha abituato (history, autocompletamento, ecc.).

NOTA:

il gdb richiede il settaggio della variabile d'ambiente `$SHELL`, cosa che non tutte le distribuzioni fanno. Prima dell'esecuzione è quindi necessario impostarla:

```
export SHELL=/bin/bash
```

In caso contrario al momento dell'esecuzione del binario il gdb restituirà l'errore:

```
Cannot exec : No such file or directory.  
Program exited with code 0177.  
You can't do that without a process to debug.
```

Comandi generici

I comandi disponibili sono molti. Esiste quindi un completo help in linea richiamabile tramite:

```
h|help  
help nomecomando  
help categoriadicomandi
```

Si noti che la maggior parte dei comandi di uso più frequente è abbreviabile utilizzando i primi caratteri del comando completo, fino alla quantità necessaria ad evitare ambiguità (Es: `q` per `quit`, `adv` per `advance`).

Inoltre in tutti i comandi in cui sia necessario specificare un punto del codice ciò viene fatto tramite una delle sintassi alternative:

```
xxxx          : numero di linea del file corrente
nomefunzione  : inizione della funzione specificata
file:xxxx     : numero di linea del file specificato
file:funzione : funzione del file specificato (in caso di ambiguità)
*xxxxxxx     : indirizzo
```

Nel caso in cui il binario da esaminare non sia stato specificato all'avvio del gdb, oppure si voglia cambiare durante la sessione:

```
file path : carica il binario specificato
```

Una tecnica molto utile consiste nell'agganciare il codice in esecuzione ad un terminale esterno, in modo che l'output sia visualizzato a parte. Per fare ciò si usa il comando:

```
tty device
```

dove con *device* si specifica il terminale desiderato (Es: /dev/pts/3, /dev/tty5, ecc.)

Infine il comando

```
quit
```

termina l'esecuzione del debugger.

Comandi di controllo dell'esecuzione

L'avvio dell'esecuzione avviene, specificando anche gli eventuali parametri di avvio, tramite le istruzioni:

```
run [argomenti] : esegue fino all'eventuale primo breakpoint
start [argomenti] : carica il programma e si arresta immediatamente
```

Per la gestione dei breakpoint, cioè dei punti di interruzione forzata dell'esecuzione, sono disponibili i comandi:

```
break punto : imposta un breakpoint al punto specificato
info break  : lista i breakpoint impostati
clear punto : cancella il breakpoint al punto specificato
```

Durante l'esecuzione sono disponibili diversi comandi per il controllo del flusso del programma. I più utili sono:

```
next          : esegue un'istruzione senza entrare in sotto-funzioni
step          : esegue un'istruzione entrando in eventuali sotto-funzioni
finish        : completa l'esecuzione dello stack-frame corrente
continue      : continua l'esecuzione fino al prossimo breakpoint
kill          : arresta l'esecuzione
advance punto : continua l'esecuzione fino al punto specificato
jump punto    : salta al punto specificato
call funzione : esegue la chiamata della funzione specificata
```

La gestione di dati e file

Per dare un'occhiata al codice durante il debugging si utilizza il comando:

```
list punto : lista 10 linee di sorgente intorno al punto specificato
             senza argomenti lista ulteriori 10 linee
```

Per il controllo della posizione di esecuzione nel filesystem:

```
cd path : analogo al comando di shell
pwd     : analogo al comando di shell
```

Per l'analisi dello stack:

```
bt|backtrace : lista i frame dello stack
frame num    : visualizza lo stack-frame num
```

Per la valutazione dei simboli (variabili&C) utilizzati nel codice:

```
print simbolo           : visualizza il valore attuale del simbolo
printf formato,simbolo,... : visualizza formattato, come la printf della stdlib
display simbolo        : come print ma continua a visualizzarne il valore
                        dopo ogni step di esecuzione
undisplay              : cancella la direttiva display
whatis simbolo         : stampa il tipo del simbolo richiesto
set VAR=EXPR          : valuta l'espressione e ne assegna il valore a VAR
```

Analisi di un core dump

Quando l'esecuzione di un programma si arresta in errore può venire generato un *core dump*, ovvero viene salvata su file l'immagine della memoria del processo al momento dell'errore.

Questa caratteristica non è normalmente abilitata, per abilitarla o disabilitarla si utilizzano nella shell i comandi:

```
ulimit -c unlimited : abilita core dump illimitati
ulimit -c 0         : disabilita core dump
```

Il core dump può quindi venire analizzato tramite il gdb:

```
gdb nomebinario core : avvia il gcc sul core-dump (core è il nome di default)
```

che, dopo aver caricato il file si troverà nello stato immediatamente precedente l'interruzione. Con i soliti comandi (*bt* e *print* sono i più utili) sarà quindi possibile analizzare il motivo e la posizione dell'errore.