.

# Debugger

Stefano Ferrari

Università degli Studi di Milano
stefano.ferrari@unimi.it

**Programmazione**

anno accademico 2017–2018

## Errori sintattici e semantici

Gli errori commessi in un programma si dividono in

- errori sintattici: riguardano la forma del programma
  - forma dei singoli termini (parole chiave, stringhe, variabili, ...)
    Es.: `flot f = 12,43; strcpy(s,"c:\prova.txt);`
  - regole di composizione delle espressioni (operatori-operandi, ...)
    Es.: `int i; i = "prova.txt"; 5 = i;`
- errori semantici: riguardano il significato del programma
  - operazioni con risultato indeterminato
    Es.: `int i, j; i = 0; j = 2 / i;`
  - operazioni con risultato diverso da quello desiderato
    Es.: `if (i = 1) printf("i è unitario");`

Il compilatore segnala gli errori sintattici

Mette solo in guardia contro alcuni possibili errori semantici

Es.: `if (i = 1) ...` `assignment used as truth value`
`i;` `statement with no effect`

.

## Debugging

Se un programma è scorretto

- sintatticamente, non viene compilato: l'eseguibile non si crea
- semanticamente, viene compilato, ma produce un eseguibile che
  - dà errore in esecuzione (*run-time error*)
  - termina fornendo il risultato sbagliato

Debugger è un programma progettato per aiutare la ricerca e la correzione degli errori semantici di programmazione

- consente l'esecuzione passo per passo del programma
- consente l'interruzione dell'esecuzione
  - in posizioni date del programma (breakpoint)
  - ogni volta che determinate espressioni cambiano valore (watchpoint)
- consente la lettura e modifica delle variabili del programma
- consente la visualizzazione della pila di sistema del programma

## GDB

GDB (GNU debugger)

- è fornito col compilatore GCC
- permette di analizzare programmi scritti in C, C++, Ada e Fortran

Gli esempi seguenti si riferiscono tutti al codice `sort.c` allegato

Prima di avviare GDB, bisogna compilare il codice con l'opzione `-g`

```
gcc -g sort.c -o sort.exe
```

Ora si può lanciare il comando `gdb`, avviando l'ambiente di debugging, come si vede dal nuovo "prompt" sul terminale

```
(gdb)
```

GDB offre una "shell" interattiva testuale:
il comando `help` mostra i comandi disponibili

```
(gdb) help
```

.

## Gestione dell'eseguibile e dei suoi parametri

Il comando `file` indica quale eseguibile interessa debuggare

```
(gdb) file sort
```

se non è stato già indicato all'avvio di GDB (o se si vuole cambiarlo)

```
gdb sort
```

Il comando `run`, seguito dai suoi eventuali parametri, lancia l'eseguibile

```
(gdb) run 3 5 4 2 1
```

oppure il comando `set arg` specifica i parametri a priori per poter poi
lanciare l'eseguibile con parametri impliciti

```
(gdb) set arg 3 5 4 2 1
(gdb) run
```

In ambiente Unix si può usare la redirezione (non in ambiente Windows!)

## Uscita del programma

Se il programma funziona, stampa a video i risultati

```
V = [ 1 2 3 5 ]
Program exited normally.
```

Se non funziona, stampa informazioni utili per individuare l'errore

```
Invalid Address specified to RtlFreeHeap( 00230000,
0240FFA0 )
```

*Utili, ma non sempre chiarissime...*

Per trovare gli errori, occorre

1. poter seguire l'esecuzione
2. poterne osservare i risultati parziali

.

## Esecuzione sospesa: breakpoint

Il comando `break` seguito dalla coppia *file*:*linea* indica che il programma, una volta lanciato, dovrà fermarsi al principio della linea e del file indicati

```
(gdb) break sort.c:28
```

Quando si lancia il comando `run`, l'esecuzione si arresta dove richiesto, stampando la linea di codice, preceduta dal numero

```
(gdb) run
Breakpoint 1, main (argc=6, argv=0x232d20) at
sort.c:28
28          InterpretaLineaComando(argc,argv,&V,&n);
```

Questi punti di arresto si chiamano breakpoint:

- si possono aggiungere quanti breakpoint si vuole
- ogni volta che il programma passa per un breakpoint, si arresta

## Esecuzione sospesa: breakpoint

Il comando `break` seguito da una funzione indica che il programma, una volta lanciato, si fermerà alla prima linea esecutiva della funzione

```
(gdb) break InterpretaLineaComando
```

Il comando `continue` prosegue l'esecuzione dal breakpoint corrente

```
(gdb) continue
```

arrestandola al successivo, ancora con la stampa della linea di codice

```
Breakpoint 2, InterpretaLineaComando (argc=6,
argv=0x232d20,
    pV=0x240ff5c, pn=0x240ff58) at sort.c:45
45          if (argc < 2)
```

(*Lanciare* `run` *ricomincerebbe da capo*)

.

## Esecuzione passo per passo

Giunti a un breakpoint, il comando `step` esegue una singola istruzione

```
(gdb) step
```

e arresta subito l'esecuzione, stampando la prima linea successiva

```
57      *pn = argc-1;
```

(*Alla riga* 45 *succede la* 57 *perché la condizione dell'if era falsa*)

Il comando `next` esegue una singola linea di codice:
se la linea è una chiamata a funzione, esegue l'intera chiamata

## Differenza tra `step` e `next`

Introduciamo un terzo breakpoint

```
(gdb) break sort.c:30
(gdb) continue
```

in modo da portarci alla chiamata della funzione di ordinamento

```
Breakpoint 3, main (argc=6, argv=0x232d20) at sort.c:30
30      OrdinaVettoreInteri(V,n);
```

Ora abbiamo due sviluppi alternativi possibili

1. con il comando `step` entriamo nella funzione

   ```
   (gdb) step
   OrdinaVettoreInteri (V=0x233fd0, n=5) at sort.c:104
   104     while (n > 1)
   ```

   (*siamo anche in un nuovo scope*)
2. con il comando `next` la eseguiamo interamente

   ```
   (gdb) next
   32      StampaVettoreInteri(V,n);
   ```

.

## Differenza tra `step` e `next` (2)

Il comando `finish` arresta l'esecuzione al termine della funzione corrente

## Breakpoint condizionali

Introduciamo un breakpoint nel ciclo di stampa del vettore finale

```
(gdb) break sort.c:120
(gdb) continue
```

Ogni comando `step` fa oscillare l'esecuzione fra le linee 119 (condizione) e 120 (corpo del ciclo); eseguendo la prima, si stampa anche l'iterazione

```
120         printf("%d",V[i]);
1 119     for (i = 1; i <= n; i++)
120         printf("%d",V[i]);
2 119     for (i = 1; i <= n; i++)
...
```

.

## Breakpoint condizionali (2)

Assegnando una condizione al breakpoint, si può scegliere un'iterazione

```
(gdb) break sort.c:120 if i >= 4
(gdb) continue
```

```
120         printf("%d",V[i]);
4 119     for (i = 1; i <= n; i++)
```

## Watchpoint

Watchpoint: l'esecuzione si arresta quando un'espressione cambia valore

Ci portiamo al principio della funzione `TrovaIndiceMassimo`

```
(gdb) break TrovaIndiceMassimo
(gdb) continue
```

introduciamo un nuovo watchpoint col comando `watch` e rilanciamo

```
(gdb) watch iMax
(gdb) continue
```

L'esecuzione si arresta appena la variabile `iMax` cambia valore

```
Continuing.
Watchpoint 4:  iMax
Old value = 2008948824
New value = 1
0x401539 in TrovaIndiceMassimo (V=0x233fd0, n=6) at sort.c:81
81 for (i = 2; i <= n; i++)
```

.

## Gestione dei punti di arresto

I comandi `info breakpoints` e `info watchpoints` mostrano l'elenco dei punti di arresto fissati (di entrambi i tipi)

Il comando `delete` *n* cancella l'*n*-esimo punto di arresto

## Osservazione e modifica di variabili

`print` *variabile* stampa il valore corrente di una variabile

```
(gdb) print iMax
$1 = 37814064
```

`display` *variabile* stampa ad ogni passo il valore corrente

```
82        if (V[i] > V[iMax]) iMax = i;
1:  iMax = 2
```

(*la stampa si interrompe quando la variabile esce dallo scope*)

`set var` *variabile* = *valore* assegna il valore alla variabile

```
set var iMax = 5
```

Nel seguito, il programma utilizza e stampa il nuovo valore

`step`

```
82        for (i = 1; i <= n; i++)
1:  iMax = 5
```

.

## Pila di sistema

Il comando `backtrace` chiede di mostrare la pila di sistema

```
(gdb) backtrace
```

Il debugger elenca gli ambienti allocati sulla pila di sistema

```
backtrace
#0 TrovaIndiceMassimo (V=0x233fd0, n=4) at sort.c:82
#1 0x4015d0 in OrdinaVettoreInteri (V=0x233fd0, n=4) at
sort.c:106
#2 0x401243 in main (argc=7, argv=0x232d48) at sort.c:30
```

Il comando `frame` *n* sposta l'attenzione sull'*n*-esimo ambiente

```
(gdb) frame 1
frame 1
#1 0x4015d0 in OrdinaVettoreInteri (V=0x233fd0, n=4) at
sort.c:106
106       i = TrovaIndiceMassimo(V,n);
```

## Scorciatoie

Ogni comando può essere sostituito dalla sua iniziale o dal suo prefisso (tranne in caso di ambiguità

`step`

`s`

`st`

Premendo INVIO si ripete il comando precedente