



Marco Alberti



**Dipartimento
di Matematica
e Informatica**



**Università
degli Studi
di Ferrara**

Programmazione e Laboratorio, A.A. 2020-2021

Ultima modifica: 5 ottobre 2020

Attenzione! Questo materiale didattico è per uso personale dello studente ed è coperto da copyright.
Ne sono vietati la riproduzione e il riutilizzo anche parziale, ai sensi e per gli effetti della legge sul diritto d'autore.

Sommario

- 1 Bug
- 2 Debugger
- 3 Comandi
- 4 Integrazione con Visual Studio Code

Bug (o errori di programmazione)

- Gli errori di sintassi impediscono la compilazione del programma
- Anche i programmi sintatticamente corretti possono contenere errori, detti **bug** o **errori logici**, che causano malfunzionamenti (cioè effetti imprevisti e generalmente indesiderati) del programma eseguibile.
- Generalmente, l'utente nota la presenza di bug a causa di output diversi da quelli che si aspetta.
- Individuare e correggere i bug è un'attività complessa (soprattutto in programmi grandi), che impiega buona parte del tempo dei programmatori.

50-70%

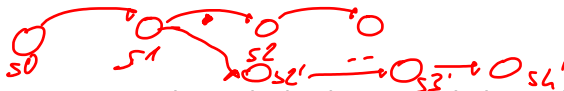
Esempio

$$\begin{array}{r} 5 \quad 7 \\ 5 + 7 = 12 \end{array}$$

Scrivere un programma che richieda all'utente due numeri interi a e b e stampi l'equazione $a + b = c$, dove c è la somma di a e b .

045_debugger/bug.c

```
1 #include <stdio.h>
2
3 main() {
4     int a, b, c;
5     scanf("%d%d", &a, &b);
6     c = a += b;
7     printf("%d + %d = %d\n", a, b, c);
8 }
```



- Se un programma stampa qualcosa di sbagliato, non è detto che l'errore sia nell'istruzione di stampa!
- In generale, gli output imprevisti (cioè la manifestazione di bug visibili agli utenti) sono originati da stati imprevisti della macchina astratta.
- In altre parole, se c'è un bug la sequenza di stati della macchina astratta sarà diversa da quella che si aspetta il programmatore.
- Esaminando gli stati della macchina astratta, il programmatore può individuare il primo che si discosta dalla sequenza prevista, e concentrarsi sulle parti del codice che l'hanno generato.

Bug

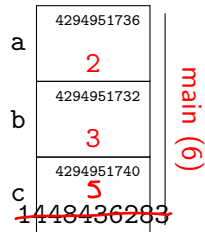
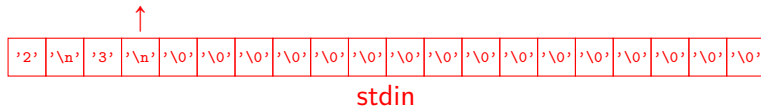
```
1  #include <stdio.h>
2
3  main() {
4      int a, b, c;
5      scanf("%d%d", &a, &b);
6      c = a += b;
7      printf("%d + %d = %d\n", a, b, c);
8  }
```

a	4294951736 -15364
b	4294951732 -15372
c	4294951740 1448436283

main (5)

Bug

```
1  #include <stdio.h>
2
3  main() {
4      int a, b, c;
5      scanf("%d%d", &a, &b);
6      c = a += b;
7      printf("%d + %d = %d\n", a, b, c);
8  }
```



Bug

errate?

BUG

```
1  #include <stdio.h>
2
3  main() {
4      int a, b, c;
5      scanf("%d%d", &a, &b);
6      c = a += b;
7      printf("%d + %d = %d\n", a, b, c);
8  }
```

↑

'2'	'\n'	'3'	'\n'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'
-----	------	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

stdin

a	4294951736 5
b	4294951732 3
c	4294951740 5

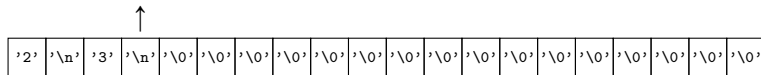
main (7)

Bug

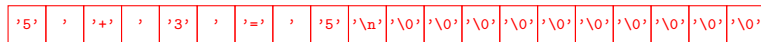
```
1  #include <stdio.h>
2
3  main() {
4      int a, b, c;
5      scanf("%d%d", &a, &b);
6      c = a += b;
7      printf("%d + %d = %d\n", a, b, c);
8  }
```

a	4294951736 5
b	4294951732 3
c	4294951740 5

main (8)



stdin



stdout

Sommario

1 Bug

2 Debugger

3 Comandi

4 Integrazione con Visual Studio Code

Che cos'è un debugger

Il **debugger** è un programma che consente di eseguire in modo controllato un altro programma, esaminando lo stato della macchina astratta e facilitando la ricerca delle cause di malfunzionamento (*bug*).

Funzionalità tipiche:

- Valutazione di espressioni
- Esecuzione passo-passo (**stepping**) -
- Interruzione in punti predefiniti (**breakpoint**)
- Interruzione in caso di modifica del valore di determinate variabili (**watchpoint**)



Il debugger per programmi in C più comune in ambiente Unix è il comando GDB (Gnu DeBugger).

Documentazione:

- Guida in linea
- `man gdb`
- `info gdb`
- *cheat sheet*: <http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>

Valutazione di espressioni

```
print __espressione__
```

visualizza il risultato della valutazione di `__espressione__`.

Esempio

p `2 + 10` visualizza il risultato dell'espressione `2 + 10`.
abbreviazione di print

Questa funzionalità consente di valutare rapidamente delle espressioni senza doverle inserire in un programma, come invece era richiesto in alcuni esercizi delle lezioni precedenti.

print 2+10 ↵

b

Compilazione con informazioni di debug

L'opzione **-g** del comando **gcc**

(eventualmente combinata con altre opzioni) inserisce nel codice generato le informazioni di debug, necessarie per esaminare il programma quando lo si esegue nel debugger.

Esempio

gcc -g __nome__.c compila con informazioni di debug il file **__nome.c**, generando l'eseguibile **a.out**.

Compilare con informazioni di debug il seguente programma.

045_debugger/bug.c

```
1  #include <stdio.h>
2
3  main() {
4      int a, b, c;
5      scanf("%d%d", &a, &b);
6      c = a += b;
7      printf("%d + %d = %d\n", a, b, c);
8  }
```


Sommario

1 Bug

2 Debugger

3 Comandi

4 Integrazione con Visual Studio Code

Avvio e chiusura del debugger

gdb

```
gdb __eseguibile__
```

apre il debugger caricando il programma `__eseguibile__`.

Ora il debugger è in attesa di comandi.

```
file __eseguibile__
```

carica il programma `__eseguibile__` (utile ad esempio se si è invocato `gdb` senza argomenti).

Esercizio

Caricare il programma compilato nell'esercizio alla slide 8.

```
quit
```

chiude il debugger.

Abbreviazioni ed help in linea

Abbreviazioni

In generale tutti i comandi di GDB ammettono qualsiasi abbreviazione non ambigua (ad esempio `quit` può essere abbreviato con `q`, `qu` o `qui`).

`help`

visualizza la guida in linea.

`run`

avvia il programma.

Esercizio

Caricare ed eseguire il programma compilato nell'esercizio alla slide 8.

Visualizzazione codice

```
list
```

visualizza le 10 righe di codice attorno alla prossima che verrà eseguita.

```
list __numero_riga__
```

visualizza le 10 righe di codice attorno a `__numero_riga__`.

Breakpoint

```
break __nome_funzione__
```

imposta un breakpoint all'inizio della funzione `__nome_funzione__` (cioè l'esecuzione sarà interrotta all'inizio dell'esecuzione di `__nome_funzione__`).

Esempio

`b main` imposta un breakpoint all'inizio del programma

break 10

```
break __numero_riga__
```

imposta un breakpoint alla riga numero `__numero_riga__`.

Esempio

`b 5` imposta un breakpoint alla riga 5.

abbreviazione di break

Breakpoint condizionali PIÙ AVANTI

```
if __condizione__
```

dove `__condizione__` è un'espressione logica, aggiunto all'impostazione di un breakpoint fa sì che l'esecuzione si interrompa solo se `__condizione__` ha valore vero.

Esempio

`b 25 if i == 5` interrompe l'esecuzione alla riga 25 se `i == 5` è vero.

`break 25 if i==5`

`break 14 if i==12`

Visualizzazione ed eliminazione breakpoint

info breakpoints

mostra i breakpoint impostati, numerati.

numero d'ordine

delete breakpoint __n__

elimina il breakpoint numero __n__.

`continue`

riprende l'esecuzione interrotta di un programma, fino al successivo breakpoint.

Esecuzione passo-passo

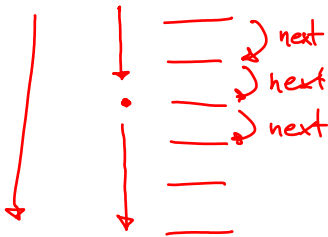
PIÙ AVANT

step

esegue una singola istruzione (eventualmente entrando nel codice delle funzioni chiamate).

next

esegue una singola istruzione (completando in un passo unico un'eventuale chiamata di funzione).



Valutazione espressioni

`print 2+3` `print a`

Abbiamo visto che

```
print __espressione__
```

visualizza il risultato della valutazione di `__espressione__`.

`__espressione__` può ovviamente contenere espressioni variabili; la valutazione utilizza il valore attuale delle variabili.

Esempio

`p a` visualizza il contenuto della variabile `a`.

`p a + 10` visualizza il risultato dell'espressione `a + 10`.

```
info locals
```

elenca, con il loro valore, tutte le variabili locali.

Esercizio

Eseguire passo-passo l'intero programma della slide 8, valutando le variabili a ogni passo.

Watchpoints

```
watch __espressione__
```

interrompe l'esecuzione ogni volta che `__espressione__` cambia di valore.

watch

Esempio

`watch a` interrompe l'esecuzione quando la variabile `a` cambia di valore.

`watch a + b` interrompe quando l'espressione `a + b` cambia di valore.

Esercizio

→ Impostare nel programma alla slide 8 un watchpoint che fermi l'esecuzione quando viene assegnata la variabile `a`. Rilevare quando `a` viene assegnata in modo inatteso.

Sommario

- 1 Bug
- 2 Debugger
- 3 Comandi
- 4 Integrazione con Visual Studio Code

Utilizzo di GDB dentro Visual Studio Code

Visual Studio Code consente il debugging di un programma con i principali comandi di GDB dalla stessa finestra di editing del codice sorgente.

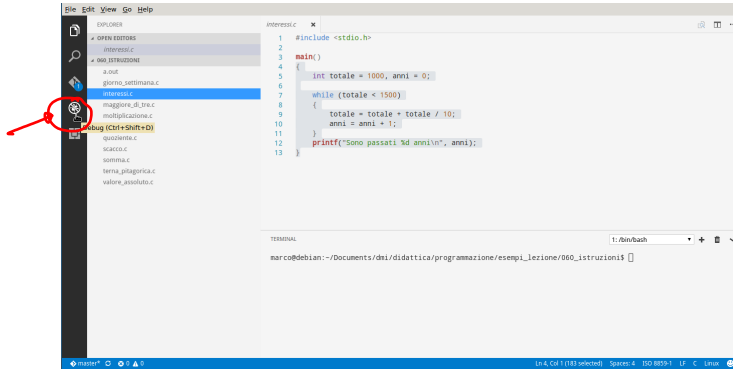
In particolare è possibile:

- ➡ • Eseguire passo-passo vedendo nel codice sorgente l'istruzione eseguita
- ➡ • Impostare breakpoint nel codice sorgente
- ➡ • Visualizzare i valori delle variabili locali di una funzione

Informazioni dettagliate: <https://code.visualstudio.com/docs/languages/cpp>.

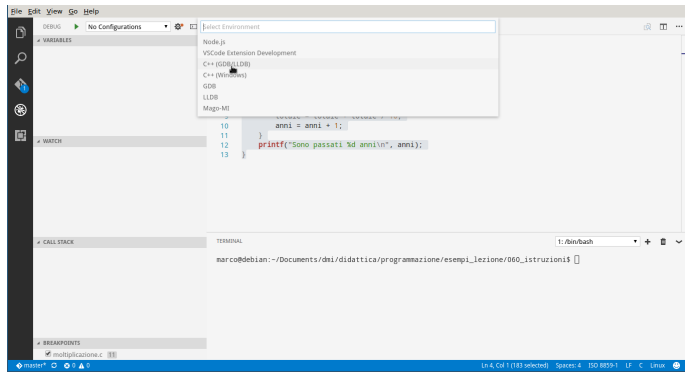
Finestra di debug

La finestra di debug si richiama con la combinazione **CTRL+SHIFT+D** o con l'apposita icona.



Impostazione di Visual Studio Code per debugging

Non è il studente / debugger
E' necessario impostare la comunicazione fra Visual Studio Code e GDB nel file `__cartella__/.vscode/launch.json`, dove `__cartella__` è la cartella di lavoro (da selezionare con il comando **Open folder** del menu **file**). Per modificare il file cliccare sull'icona dell'ingranaggio; se non esiste cliccare sull'icona stessa, poi su "C++(GDB/LLDB)", poi su "Default configuration".



Tipico contenuto del file `launch.json`

Il seguente contenuto è sufficiente per iniziare:

045_debugger/launch.json

```
1 {  
2     "version": "0.2.0",  
3     "configurations": [  
4         {  
5             "name": "C++ Launch",  
6             "type": "cppdbg",  
7             "request": "launch",  
8             "program": "${workspaceRoot}/a.out",  
9             "args": [],  
10            "stopAtEntry": false,  
11            "cwd": "${workspaceRoot}"  
12        }  
13    ]  
14 }
```

(eventualmente sostituire `a.out` con il nome dell'eseguibile)

Eeguire dentro Visual Studio Code le operazioni eseguite in GDB nei precedenti esercizi.

Notare che i watchpoint non sono supportati in Visual Studio Code.