# Eseguibile e debugging

## Il debugger

Il debugger è uno strumento software che permette di verificare l'esecuzione di altri programmi. Il suo utilizzo risulta indispensabile per trovare errori (bug, da qui il nome debugger) in programmi di complessità elevata. Le principali caratteristiche di un debugger sono:

- possibilità di eseguire il programma *passo-passo*
- possibilità di arrestare in modo condizionato l'esecuzione del programma tramite l'inserimento di punti di arresto detti *breakpoint*;
- possibilità di *visualizzare e modificare* il contenuto dei registri e della memoria.

Il debugger più diffuso in ambiente Linux è **gdb**. Il programma gdb funziona in modalità testo, pertanto i comandi vengono impartiti mediante il prompt. Tuttavia, per semplificare il suo utilizzo sono stati sviluppati numerosi front-end grafici, il più diffuso dei quali risulta essere **ddd**.
Per poter utilizzare un debugger, i programmi devono essere assemblati e linkati opportunamente tramite le seguenti righe di comando:

```
as –gstabs -o miofile.o miofile.s
ld -o miofile miofile.o
```

L'opzione `–gstabs` permette di inserire nel file oggetto, e quindi nell'eseguibile, le informazioni necessarie al debugger.

Comandi più frequenti di gdb:

| Comando | Descrizione |
|---|---|
| `file <nome_eseguibile>` | Carica il programma per il debugging. |
| `break file_sorgente:numero_riga` | Imposta un breakpoint alla riga specificata del file sorgente specificato (file avente estensione .s). Il programma si interromperà prima di eseguire tale istruzione. Per questo motivo non è possibile mettere un breakpoint sulla prima istruzione del programma. |
| `run [parametri]` | Esegue il programma. L'esecuzione del programma si sospende quando viene raggiunto il primo breakpoint. Nel caso non vi siano breakpoint l'esecuzione avviene normalmente. È possibile eventualmente specificare parametri da |

| | |
|---|---|
| | passare al programma sulla linea di comando. |
| `list` | Visualizza una porzione di file sorgente. Utilizzando ripetutamente il comando è possibile scorrere tutto il file. |
| `list <file_sorgente>` | Come il comando precedente ma cambia il file sorgente da ispezionare. Successivamente è sufficiente utilizzare ripetutamente il comando *list* per scorrere tutto il file. |
| `step` | Esegue l'istruzione corrente quando l'esecuzione del programma è sospesa a seguito del raggiungimento di un breakpoint. Reiterare il comando *step* per continuare ad eseguire un'istruzione alla volta. |
| `next` | Similmente al comando *step* esegue l'istruzione corrente, ma nel caso si tratti di una chiamata a funzione, essa viene eseguita come fosse una singola istruzione senza visualizzare le istruzioni che la compongono. |
| `continue` | Prosegue l'esecuzione del programma fino al prossimo breakpoint. |
| `finish` | Prosegue l'esecuzione del programma fino alla fine. |
| `info registers` | Visualizza il contenuto dei registri quando il programma è sospeso su un'istruzione (per un breakpoint o l'uso di next e step). |
| `p/formato $registro` | Stampa il contenuto del registro "*registro*" nel formato indicato dall'opzione "*formato*". Le possibili opzioni sono: *x* per esadecimale, *o* per ottale, *d* per decimale, *t* per binario.<br>Ad esempio per stampare il contenuto del registro EAX in binario bisogna lanciare il comando `p/t $eax`.<br>Funziona solo quando il programma è sospeso su un'istruzione (per un breakpoint o l'uso di next e step). |
| `print <nome_variabile>` | Visualizza il contenuto della variabile "*nome_variabile*".<br>Funziona solo quando il programma è sospeso su un'istruzione (per un breakpoint o l'uso di next e step). |

| | |
|---|---|
| `x/nb indirizzo` | Visualizza il contenuto di *n* byte della memoria a partire della locazione di cui viene fornito l'indirizzo. L'indirizzo può essere rappresentato dal nome di una variabile.<br>Ad esempio, il comando `x/4b &nome_variabile` visualizza il contenuto di 4 byte della memoria a partire dall'indirizzo associato alla variabile "*nome_variabile*".<br>Funziona solo quando il programma è sospeso su un'istruzione (per un breakpoint o l'uso di next e step).<br>Affinché gdb riesca a visualizzare correttamente il contenuto delle variabili in memoria con questo comando occorre che esse abbiano tutte nomi diversi anche se dichiarate in file diversi. |
| `x/nw indirizzo` | Visualizza il contenuto di *n* parole da 32 bit della memoria a partire della locazione di cui viene fornito l'indirizzo. Per gdb una parola è costituita da 4 byte. L'indirizzo può essere rappresentato dal nome di una variabile.<br>Ad esempio, il comando `x/1w &nome_variabile` visualizza il contenuto di 1 campo da 32 bit della memoria a partire dall'indirizzo associato alla variabile "*nome_variabile*". Funziona solo quando il programma è sospeso su un'istruzione (per un breakpoint o l'uso di next e step).<br>Affinché gdb riesca a visualizzare correttamente il contenuto delle variabili in memoria con questo comando occorre che esse abbiano tutte nomi diversi anche se dichiarate in file diversi. |
| `Help` | Visualizza le istruzioni per l'utilizzo della guida in linea. |

I comandi sopra elencati possono essere eseguiti anche utilizzando *ddd*. In tal caso la loro esecuzione avviene cliccando sui corrispondenti pulsanti nelle barre degli strumenti o sulle voci dei menu.

# Makefile

Tramite il comando `make` digitato direttamente nella shell è possibile compilare e linkare i file sorgenti, questo comando si basa su un file chiamato `Makefile` che contiene tutte le direttive per la compilazione e il linking.
Il makefile è composto da un insieme di target e di regole corrispondenti strutturati nella seguente maniera:

```
Target 1: lista dei file da analizzare
     Regola
Target 2: lista dei file da analizzare
     Regola
...
```

invocando semplicemente il comando `make` verrà invocato il primo target mentre lanciando il comando seguito dal nome di un target invocherà il target corrispondente.
All'interno del `makefile` si possono dichiarare variabili semplicemente scrivendone il nome in maiuscolo seguito dal simbolo "=" e dal valore che gli si vuole assegnare, per poterla poi utilizzare all'interno del programma il nome della variabile va chiuso da parentesi tonde il tutto preceduto dal carattere "`$`"; ad es. `$(NOME_VARIABILE)`.
Se uno dei file da analizzare per l'invocazione di un target è stato modificato, il comando `make` esegue il target che corrisponde alla ricompilazione di quel file.

NB: è molto importante che le istruzioni siano precedute da 1 <tab>. Attenzione a non usare spazi ma solo e soltanto il tasto <tab>.

NB: per poter funzionare il makefile deve chiamarsi `makefile` e deve essere salvato nella cartella che contiene tutti i file necessari alla compilazione, per ulteriori informazioni digitare il comando `man make`.

Esempio di makefile:

```
AS = as --32
LD = ld -m elf_i386
FLAGS = -gstabs

all:
     $(AS) $(FLAGS) -o nomeSorgente.o nomeSorgente.s
     $(LD) -o nomeSorgente.x nomeSorgente.o
clean:
     rm -f nomeSorgente.o nomeSorgente.x core
```

# Esercizi

## Esercizio 1

Scrivere un programma Assembly equivalente al seguente algoritmo C per il calcolo del massimo comune divisore (MCD) di due numeri naturali. Si assuma che gli interi `a` e `b` siano nei registri EAX ed EBX, mentre il risultato deve essere inizialmente memorizzato in ECX, convertito in testo e stampato a video.
Verificarne il funzionamento con il debugger.

```
#include <stdio.h>

int main(void){

     int a = 25; // da salvare in EAX
     int b = 30; // da salvare in EBX
     int result; // da salvare in ECX

     if (a==0 && b==0)
          result = 1;
     else if (a==0)
          result = b;
     else if (b==0)
          result = a;
     else {
          while (a!=b)
               if (a<b)
                    b = b - a;
               else
                    a = a - b;
          result = a; // MCD = a = b
     }

     printf("PROGRAMMA PER IL CALCOLO DEL MCD\n\n");
     printf("Valore richiesto: %i\n", result);

    // termino ritornando zero (chiusura senza errori)
    return 0;
}
```

## Esercizio 2

Scrivere un programma Assembly che calcoli il fattoriale di un numero naturale e ne stampi il risultato a video. La gestione dei registri nel programma deve essere concepita in modo tale da rendere possibile il calcolo di 12! = 479.001.600