C++ Programming

Gianluigi Ciocca, Luigi Celona

g++
Compilazione e linking

- *g++* consente di effettuare la compilazione e il linking di codice *C++*

- Compilazione: si usa il parametro *-c*
  - `g++ -c main.cpp`

  Viene così generato un file di **codice oggetto** chiamato *main.o*

- Linking: è sufficiente fornire il codice oggetto e specificare con il parametro *-o* (che sta per output) il nome dell'eseguibile
  - `g++ main.o -o test`

  In questo caso l'eseguibile generato avrà nome *test*. Sotto windows-mingv dovete specificare anche l'estensione - *test.exe*

  Se non viene specificato il parametro *-o* viene creato di default un eseguibile chiamato *a.out*

g++
Includes

- Per specificare le directory di inclusione dei file header si usa l'opzione *-I* ("i" maiuscola)

```
g++ -I./include:/pippo/headers -c main.cpp
```

- In questo esempio compila il file *main.cpp* cercando eventuali headers nelle directory *./include* e */pippo/headers*. Il simbolo ":" serve a separare diversi percorsi

g++
Esempio di progetto multi-file

- Progetto con una directory *src* e una *include*; 2 header e due file *.cpp*:
  - *./main.cpp*
  - *./include/Point.h*
  - *./include/Rectangle.h*
  - *./src/Point.cpp*
  - *./src/Rectangle.cpp*
- Per compilare e linkare, supponendo di essere nella directory *./* (dove si trova *main.cpp*):
  - `g++ -c -Iinclude main.cpp -o main.o`
  - `g++ -c -Iinclude src/Point.cpp -o Point.o` **(NOTA: genera *Point.o* nella directory *./* e non in *./src*)**
  - `g++ -c -Iinclude src/Rectangle.cpp -o Rectangle.o` **(NOTA: genera *Rectangle.o* nella directory *./* e non in *./src*)**
  - `g++ -o main main.o Point.o Rectangle.o`

g++
Esempio di progetto multi-file

- Progetto con una directory *src* e una *include*; 2 header e due file *.cpp*:
  - *./main.cpp*
  - *./include/Point.h*
  - *./include/Rectangle.h*
  - *./src/Point.cpp*
  - *./src/Rectangle.cpp*
- Per compilare e linkare, supponendo di essere nella directory *./* (dove si trova *main.cpp*): **compilazione**
  - `g++ -c -Iinclude main.cpp -o main.o`
  - `g++ -c -Iinclude src/Point.cpp -o Point.o` **(NOTA: genera *Point.o* nella directory *./* e non in *./src*)**
  - `g++ -c -Iinclude src/Rectangle.cpp -o Rectangle.o` **(NOTA: genera *Rectangle.o* nella directory *./* e non in *./src*)**
  - `g++ -o main main.o Point.o Rectangle.o`

g++
Esempio di progetto multi-file

- Progetto con una directory *src* e una *include*; 2 header e due file *.cpp*:
  - *./main.cpp*
  - *./include/Point.h*
  - *./include/Rectangle.h*
  - *./src/Point.cpp*
  - *./src/Rectangle.cpp*
- Per compilare e linkare, supponendo di essere nella directory *./* (dove si trova *main.cpp*):
  - `g++ -c -Iinclude main.cpp -o main.o`
  - `g++ -c -Iinclude src/Point.cpp -o Point.o` **(NOTA: genera *Point.o* nella directory *./* e non in *./src*)**
  - `g++ -c -Iinclude src/Rectangle.cpp -o Rectangle.o` **(NOTA: genera *Rectangle.o* nella directory *./* e non in *./src*)**
  - `g++ -o main main.o Point.o Rectangle.o`

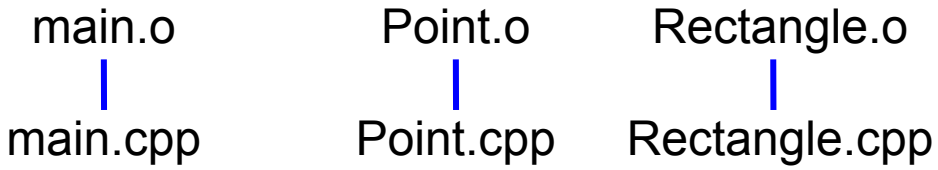
linking

Grafo delle dipendenze
Cos'è

- Metodo di rappresentazione delle dipendenze che intercorrono tra i vari sorgenti/prodotti/sottoprodotti della compilazione

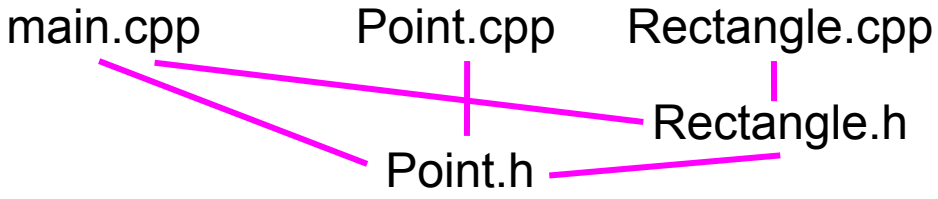
Dipendenze per il linking

Dipendenze di compilazione



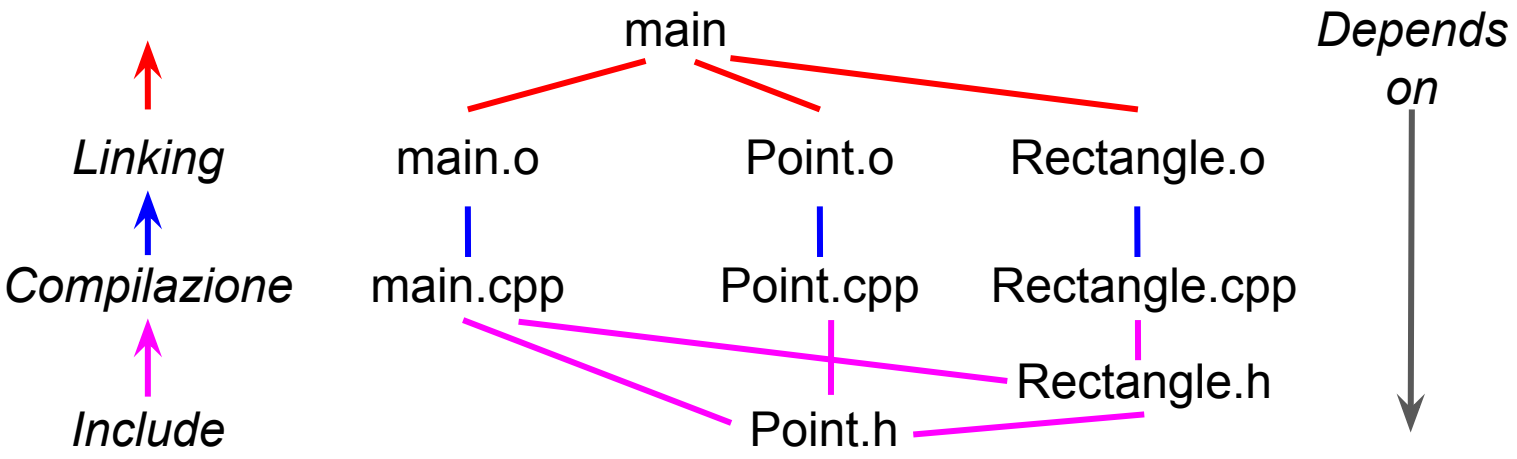
Dipendenze di inclusione

NOTA: tali dipendenze hanno un altro significato poiché non sono utilizzate per *generare* altri file a differenza dei file **.cpp* e **.o*



Grafo delle dipendenze
Cos'è

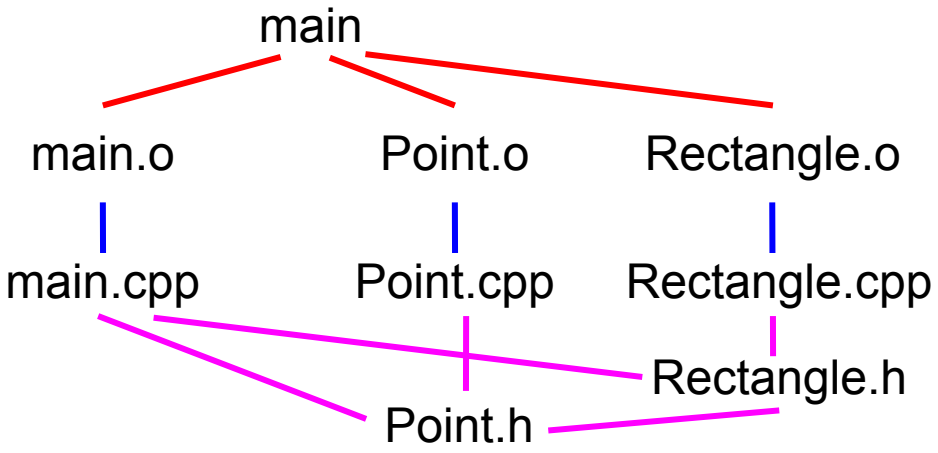
- Metodo di rappresentazione delle dipendenze che intercorrono tra i vari sorgenti/prodotti/sottoprodotti della compilazione



NOTA: potrebbero essere presenti dipendenze **indirette** (come quella tra *Rectangle.cpp* e *Point.h*, poiché *Rectangle.cpp* include *Rectangle.h*, che include *Point.h*)

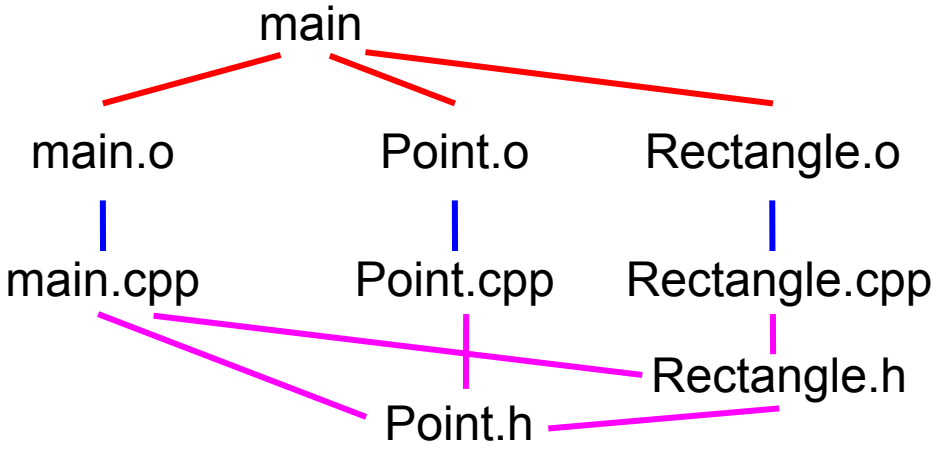
Grafo delle dipendenze
A cosa serve

- Per poter comprendere quali parti devono essere rigenerate dopo aver modificato specifici moduli del programma
  - **Domanda 1:** Modificando il file *main.cpp*, cosa deve essere rigenerato?



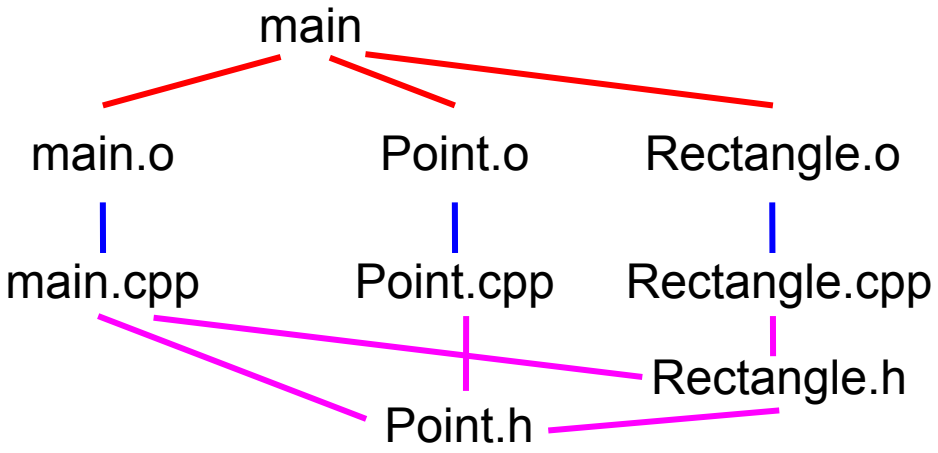
Grafo delle dipendenze
A cosa serve

- Per poter comprendere quali parti devono essere rigenerate dopo aver modificato specifici moduli del programma
  - **Domanda 1:** Modificando il file *main.cpp*, cosa deve essere rigenerato?
  - **Risposta:** Prima ricompilare il file *main.cpp* per ottenere il file oggetto *main.o*, dopo re-linking di tutti i file oggetto per ottenere l'eseguibile *main*.



Grafo delle dipendenze
A cosa serve

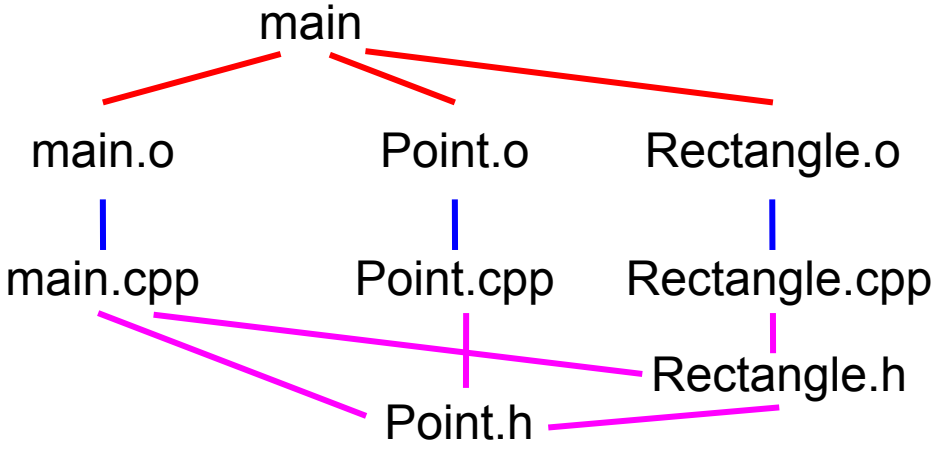
- Per poter comprendere quali parti devono essere rigenerate dopo aver modificato specifici moduli del programma
  - **Domanda 1:** Modificando il file *main.cpp*, cosa deve essere rigenerato?
  - **Risposta:** Prima ricompilare il file *main.cpp* per ottenere il file oggetto *main.o*, dopo re-linking di tutti i file oggetto per ottenere l'eseguibile *main*.
  - **Domanda 2:** Se invece modificassimo il file *Point.h*?



Grafo delle dipendenze
A cosa serve

- Per poter comprendere quali parti devono essere rigenerate dopo aver modificato specifici moduli del programma
  - **Domanda 1:** Modificando il file *main.cpp*, cosa deve essere rigenerato?
  - **Risposta:** Prima ricompilare il file *main.cpp* per ottenere il file oggetto *main.o*, dopo re-linking di tutti i file oggetto per ottenere l'eseguibile *main*.
  - **Domanda 2:** Se invece modificassimo il file *Point.h*?
  - **Risposta:** Poiché tutti i sorgenti dipendono dall'header file *Point.h*, è necessario **ricompilare tutti i file** ed effettuare il re-linking di tutti i file oggetto per ottenere l'eseguibile *main*.

NOTA: Quando un file **.cpp* **o un header file in esso incluso** viene modificato è necessario rigenerare il corrispondente file oggetto



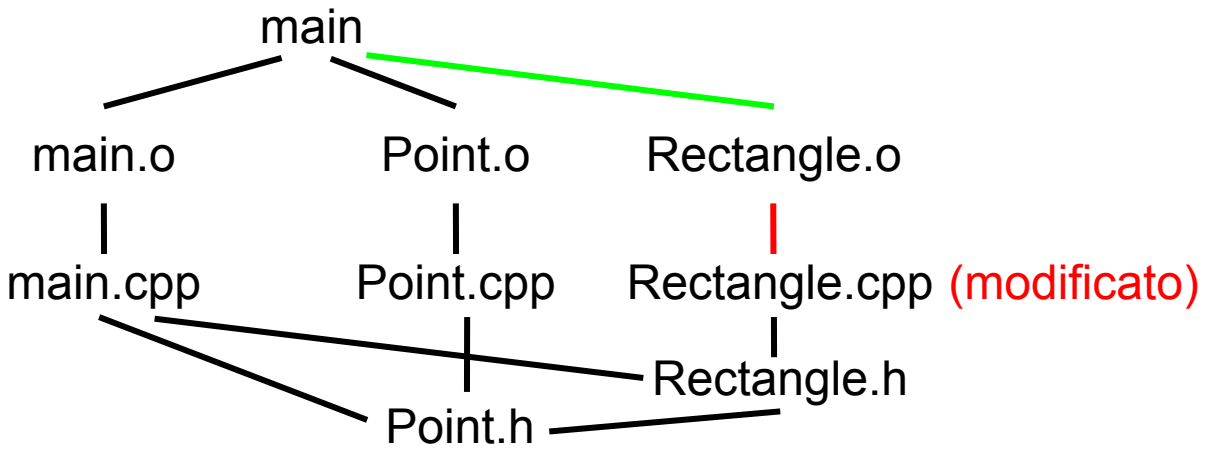
Make
Cos'è e a cosa serve

- Strumento per la compilazione automatica di codice sorgente (C/C++, LaTeX, ...)
- Disponibile su più piattaforme (Unix, Windows, OSX)
- Fondamentale per progetti composti da molti file sorgenti
- Ottimizza il processo di ricompilazione, considerando esclusivamente i file sorgenti modificati
- Richiede un **Makefile** per codificare le dipendenze tra i sorgenti e i comandi utili per la generazione dei file
    - In sostanza si tratta di una rappresentazione del grafo delle dipendenze

NOTA: ***make*** è il programma da eseguire, ***Makefile*** è invece il file con cui viene specificato a *make* come compilare/linkare i file sorgenti

Makefile
Grafo delle dipendenze

- Quando *make* viene invocato
  - Viene letto il contenuto del *Makefile* e vengono eseguiti i passi necessari per generare i file
  - Contemporaneamente viene verificata la data di modifica dei sorgenti e determina cosa deve essere rigenerato. I file **più vecchi** delle proprie dipendenze **devono** essere rigenerati (la rigenerazione di un file potrebbe causare una reazione a catena)



Makefile
Regole

- Un *Makefile* è un insieme di regole che descrivono i rami dell'albero di dipendenze. Ogni regola è così composta:

```
target: dipendenza_1 dipendenza_2 … dipendenza_N
<TAB> azione_1
<TAB> azione_2
…
<TAB> azione_M
```

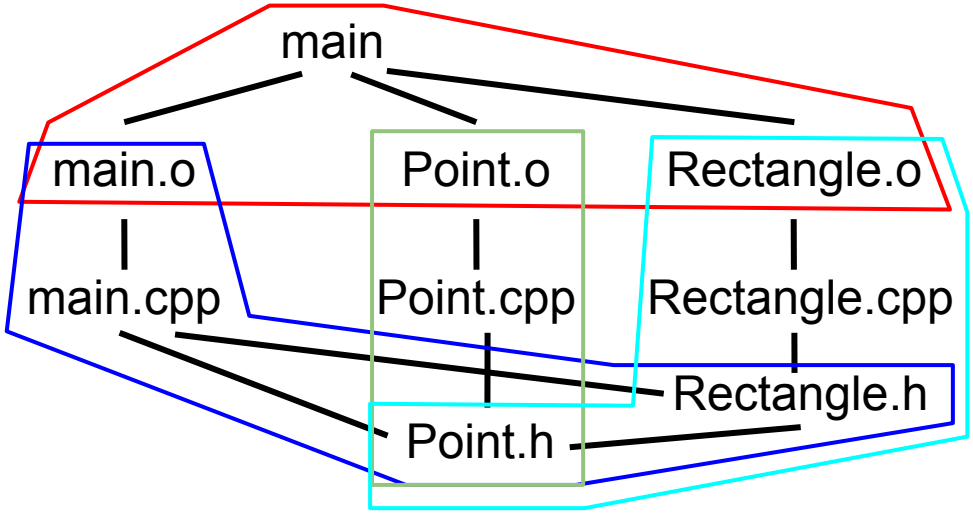
- Il **target** e le **dipendenze** sono nomi di file, mentre le **azioni** sono dei comandi di shell che *make* va ad eseguire per generare il target a partire dalle dipendenze.

Makefile
Tabulazione!

- Assicuratevi che il vostro editor di testo inserisca dei simboli di tabulazione *\t* quando premete il tasto “tab” (specie se lavorate in ambiente Windows)
- Le azioni devono essere precedute da tab, altrimenti *make* lo considera un errore di sintassi!

```
Makefile:2: *** missing separator. Stop.
```

Makefile
Esempio




```
main: Point.o Rectangle.o main.o
	g++ -o main Point.o Rectangle.o main.o

main.o: main.cpp
	g++ -I./include -c main.cpp

Point.o: ./src/Point.cpp
	g++ -I./include -c ./src/Point.cpp

Rectangle.o: ./src/Rectangle.cpp
	g++ -I./include -c ./src/Rectangle.cpp
```

- *Make* se lanciato senza parametri cerca un file di nome *Makefile* (con la M maiuscola)
- *Make* cerca di portare a termine il target della prima regola che incontra nel *Makefile* (se non specificato altrimenti)

Makefile
Variabili

- È possibile definire delle variabili che possono essere poi richiamate all'interno del file
- Utili per memorizzare il valore di alcuni parametri, ad esempio:
  - Il tipo di compilatore da utilizzare: `CXX = g++`
  - Flag da passare al compilatore C++: `CXXFLAGS = -Wall -g`
  - Opzioni per il linker: `LDFLAGS = -Wl`
- Per sostituire il valore di una variabile al nome si utilizza il segno ($) e il nome della variabile tra parentesi `$(CXX)`

```
CXX = g++
INCLUDES = -I./include
```

```
main: Point.o Rectangle.o main.o
	g++ -o main Point.o Rectangle.o main.o

main.o: main.cpp
	g++ -I./include -c main.cpp

Point.o: ./src/Point.cpp
	g++ -I./include -c ./src/Point.cpp

Rectangle.o: ./src/Rectangle.cpp
	g++ -I./include -c ./src/Rectangle.cpp
```

Makefile
Variabili automatiche

Automatic Variable	Informazione
$@	Nome del target
$%	Nome del target, che è un membro di un archivio
$^	Nomi di tutte le dipendenze, separate da spazi
$<	Nome della prima dipendenza
$?	Nomi di tutte le dipendenze più recenti del target
$+	Nomi di tutte le dipendenze con duplicati in ordine

```
main: Point.o Rectangle.o main.o
    $(CXX) $(CXXFLAGS) -o $@ $^
```

- In questo esempio $@ è uguale a “main”, mentre $^ equivale a “Point.o Rectangle.o main.o”

Makefile
VPATH

- Nel caso di progetti complessi in cui i file sono dislocati in diverse directory, è possibile specificare i percorsi nella variabile VPATH

```
VPATH=./src:./include
```

- I file legati a target e dipendenze verranno cercati nelle directory specificate da VPATH nel caso in cui non venissero trovati nella directory corrente

Makefile
PHONY

- A volte è utile specificare dei target che non sono associati a nessun file “reale”. È possibile specificare tali target tramite la direttiva **.PHONY**
- Ad esempio spesso viene definito il target **clean** per eliminare tutti i prodotti e sottoprodotti del processo di compilazione, in modo da ripartire dalla situazione iniziale

```
.PHONY: clean
clean:
    rm -rf *.o main
```

Make
Come invocarlo

- Sintassi: `make [options] [targets]`
- Se invocato senza parametri *make* cerca il *makefile* di nome “Makefile” e ne esegue la prima regola
- Per specificare un *makefile* diverso
  - `make -f` *`nomefile`*
- È possibile specificare il target da soddisfare al posto del primo trovato
  - Ad esempio, `make clean`
- Per far si che *make* non si arresti al primo errore ma prosegua nell’albero delle dipendenze
  - `make -k`

Debugging
GDB

- **G**nu **D**e**B**ugger: è un debugger a linea di comando
- È necessario utilizzare come opzioni di compilazione “-g -O0” nel caso in cui si voglia effettuare debugging con gdb
- Per invocare GDB sull’eseguibile *main*
  - `gdb main`

GDB
Comandi utili: esecuzione

- **help**: visualizza i comandi disponibili e la relativa documentazione
- **start**: lancia il programma caricato con i parametri specificati e si ferma nell'entry point
  - Esempio: `start -c config.cfg`
- **r** (run): esegue il programma per intero e termina
- **n** (next): esegue l'istruzione successiva e si ferma. Se si tratta di una chiamata a funzione, la esegue completamente senza entrare nel corpo
- **s** (step): esegue l'istruzione successiva e si ferma. Se si tratta di una chiamata a funzione, entra nel corpo della stessa
- **l** (list): stampa la porzione di codice che è attualmente interessata, oppure la funzione o il numero di riga specificato
- **q** (quit): esce da gdb
- **k** (kill): termina l'esecuzione corrente
- **u** (util): esegue il programma fino alla riga di codice specificata

Nota: premendo semplicemente "Enter", viene ripetuto l'ultimo comando!

GDB
Variabili, stack e memoria

- **p** (print): stampa il valore di una variabile o di una espression
- **bt** (**b**ack**t**race) e where: stampa la traccia dello stack
- i (info): stampa informazioni in base all'argomento passato, ad esempio:
  - **i local**: Variabili locali
  - **i variables**: Tutte le variabili
  - **i args**: Argomenti passati allo stack frame corrente
  - **i breakpoints**: Breakpoints: breackpoints definiti
  - ...
- **display:** stampa un'espressione (come "p") ogni volta che il programma si ferma
- **undisplay:** annulla un comando display

GDB
Breakpoints

- **b** (break): permette di specificare un breakpoint secondo diverse modalità:
  - `b` *`nome_funzione`*
  - `b` *`nome_file.cpp:<numero_riga>`*
- **condition**: permette di aggiungere una condizione ad un breakpoint già definito
  - `condition <id_breakpoint> condizione`

  Esempio:

  ```
  condition 1 i==6
  ```

- **delete**: per eliminare un breakpoint
  - `delete <id_breakpoint>`