UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Corso di Laurea in Informatica *(Track B)* - A.A. 2018/2019

# Laboratorio di Informatica

## Debugging

**docente: Veronica Rossano**

veronica.rossano@uniba.it

Slides ispirate ai contenuti proposti dal prof. Corrado Mencar. Grazie.

## Debugging

- **Debug** = processo di riconoscimento e rimozione dei bug



## Debugging

- **Debug** = processo di riconoscimento e rimozione dei bug
- **Bug** = errore presente nel software
  - **Errori Sintattici:**
  - **Errori Semantici:**



## Debugging

- **Debug** = processo di riconoscimento e rimozione dei bug
- **Bug** = errore presente nel software
  - **Errori Sintattici:** rilevati sempre dal compilatore in fase di compilazione
    - **Esempio:** variabili non dichiarate, assenza del ';' a fine istruzione, etc.
  - **Errori Semantici:** difficilmente rilevabili
    - **Esempio:** uso errato delle parentesi, contatori utilizzati in modo errato, confusione tra = e ==, etc.

## Debugging

- **Debug** = processo di riconoscimento e rimozione dei bug
- **Bug** = errore presente nel software
  - **Errori Sintattici:** rilevati sempre dal compilatore in fase di compilazione
    - **Esempio:** variabili non dichiarate, assenza del ';' a fine istruzione, etc.
  - **Errori Semantici:** difficilmente rilevabili
    - **Esempio:** uso errato delle parentesi, contatori utilizzati in modo errato, confusione tra = e ==, etc.
- **Attenzione**:
  - I bug sono molto frequenti, anche in programmi semplici
  - Il debug è un'attività difficile, che richiede un tempo imprevedibile
  - Occorre adottare tutte le tecniche che **riducano la presenza di bug** e il **tempo del debug**
  - **Più è grande il programma, più è difficile trovare gli errori**



## Debugging

*Debugging **is twice as hard** as writing the code in the first place*

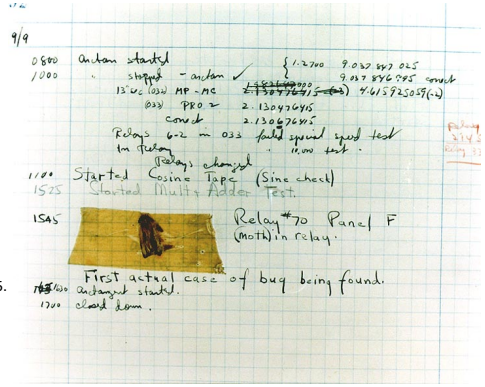
**Brian Kernighan**



## Debugging: **storia**

Il 9 settembre 1947 il tenente Grace Hopper ed il suo gruppo stavano cercando la causa del malfunzionamento di un computer Mark II quando, con stupore, si accorsero che **una falena si era incastrata tra i circuiti**. Dopo aver rimosso l'insetto (alle ore 15.45), il tenente incollò la falena rimossa sul registro del computer e annotò: «1545. Relay #70 Panel F (moth) in relay. **First actual case of bug being found**».

*fonte: Wikipedia*



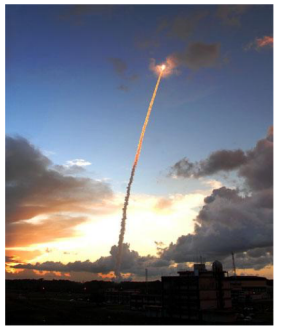



## Debugging: **perché?**

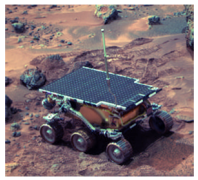
(1996)
**Thirty-six seconds into** its maiden launch the rocket's engineers hit the self destruct button following multiple computer failures.

In essence, **the software had tried to cram a 64-bit number into a 16-bit space**. The resulting overflow conditions crashed both the primary and backup computers (which were both running the exact same software).

The Ariane 5 had cost nearly **$8 billion to develop**, and was carrying a **$500 million** satellite payload when it exploded.



## Debugging: perché?



*"The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. [...] But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data."*
(D. Wilner, 1997 IEEE Real-Time Systems Symposium)

**Se un bug è individuato, va eliminato subito.** Il trasferimento di un bug nei passi successivi del ciclo di sviluppo di un software **fa crescere il costo del debugging in termini esponenziali.**



## Debugging: come?

- **Bug** = errore presente nel software
  - **Errori Sintattici:** rilevati sempre dal compilatore in fase di compilazione
    - **Esempio:** variabili non dichiarate, assenza del ';' a fine istruzione, etc.
  - **Errori Semantici:** difficilmente rilevabili
    - **Esempio:** uso errato delle parentesi, contatori utilizzati in modo errato, confusione tra = e ==, etc.
- Il debugging è ovviamente focalizzato sulla **rimozione degli errori semantici**
- **Che tipologia di errori (semantici)** possiamo incontrare?



## Debugging: come?

- Il debugging è ovviamente focalizzato sulla **rimozione degli errori semantici**
- **Che tipologia di errori (semantici)** possiamo incontrare?
  - **Interruzione inattesa** del programma
  - Il programma **non si ferma** più
  - Il programma termina dando **risultati sbagliati**



## Debugging: come?

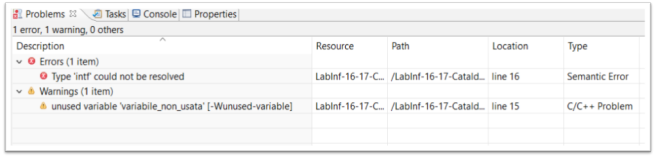
- Il debug di un programma consta di tre fasi successive:
  1. **trovare le istruzioni** che causano il bug
  2. **scoprire il motivo** del bug
  3. **correggere** il codice
- La prima fase è certamente la più difficile **e le tecniche** da utilizzare nella individuazione dei bug **dipendono dalla tipologia di errori (semantici)**
  - Prima di adottare il debugger, **esistono delle linee guida / accorgimenti che è bene seguire** per individuare le istruzioni che causano il bug

## 1) Supporto del compilatore

- Molti compilatori **emettono dei "warning",** cioé dei messaggi di avvertimento
  - `if (a=0) …`
  - `x = x`
  - nessun return
- Analizzare attentamente i warning emessi dal compilatore. **Molto spesso dietro un warning può nascondersi un potenziale bug.**



## 1) Supporto del compilatore




Esempio di Warning in **Eclipse CDT**



## 2) Pattern familiari

- **Riconoscere variazioni** rispetto a "modelli" (pattern) di codice familiari

```
int n;
scanf("%d", n);
```

```
int n;
scanf("%d", &n);
```

- **Consiglio:** L'uso di un **corretto stile di programmazione** aiuta a ridurre la presenza di bug



## 3) Esaminare codice simile

- Se un bug è presente in una porzione di codice, allora è probabile che se ne annidi un **altro in un codice simile**
  - problema del *"copy-and-paste"*
  - Es) Tipicamente avviene nei cicli, che hanno spesso una struttura standard. **Ad esempio se si sbaglia la condizione di uscita**

## 3) Esaminare codice simile

- Se un bug è presente in una porzione di codice, allora è probabile che se ne annidi un **altro in un codice simile**
  - problema del *"copy-and-paste"*
  - Es) Tipicamente avviene nei cicli, che hanno spesso una struttura standard. **Ad esempio se si sbaglia la condizione di uscita**
- **Una buona progettazione del codice riduce la ridondanza** e, quindi, la possibilità di bug duplicati
  - Porzioni di codice che svolgono operazioni simili **possono essere codificate attraverso funzioni o procedure**. In tal caso il bug si presenterà solo una volta, tipicamente dentro la funzione.



## 4) Backward reasoning

- Quando si scopre un bug, occorre "pensare al contrario"
  - Partendo dal risultato, occorre risalire alla catena delle cause che lo hanno portato. Una delle cause della catena sarà errata



## 4) Backward reasoning

- Quando si scopre un bug, occorre "pensare al contrario"
  - Partendo dal risultato, occorre risalire alla catena delle cause che lo hanno portato. Una delle cause della catena sarà errata
  - Es.) Ho prodotto un risultato. In che variabile è contenuto il risultato? Quali istruzioni hanno modificato quella variabile? L'**errore sarà certamente in una di quelle istruzioni**
  - Es.) Se appare un bug **ogni qual volta viene invocata una funzione**, probabilmente l'errore è dentro la funzione
- Scrivere **codice leggibile** aiuta il **backward reasoning** e, quindi, a localizzare i bug



## 5) Sviluppo incrementale

- Testare le procedure man mano che vengono sviluppate
  - Se i test all'istante **t** hanno **successo** ma **falliscono** all'istante **t+1**, allora molto probabilmente i bug si annidano nel codice sviluppato tra **t** e **t+1**

## 5) Sviluppo incrementale

- Testare le procedure man mano che vengono sviluppate
  - Se i test all'istante **t** hanno **successo** ma **falliscono** all'istante **t+1**, allora molto probabilmente i bug si annidano nel codice sviluppato tra **t** e **t+1**
  - **Esempio**
    - Il valore delle variabili prima di un ciclo è quello atteso, ma dopo il ciclo il valore non è più corretto. **Allora è chiaro che il bug è localizzato dentro il ciclo.**
    - Il valore di una variabile prima di una funzione è quello atteso, dopo la funzione non è più corretto. **Allora è chiaro che il bug è stato provocato dalla funzione.**
- **La progettazione modulare** del codice aiuta a individuare meglio la posizione dei bug



## 6) Leggere e spiegare il codice

- Leggere il codice e comprenderne il significato
  - Il codice è un frammento di «conoscenza» che deve essere compreso sia dalla macchina che da chi la programma
  - **La leggibilità del codice è fondamentale**
    - **Difficoltà a spiegare o commentare un pezzo di codice sono probabilmente indice di una esagerata complessità, che a sua volta è indice di potenziali bug**
- **Consiglio:** Spiegare ad altri il codice aiuta a ridurre problemi
  - Assicuratevi che il codice sia sempre comprensibile, provando a spiegarlo ad altri



## 7) Rendere riproducibile un bug

- Individuare tutte le condizioni che portano alla manifestazione di un bug
  - Input e altri parametri
  - Condizioni della macchina
  - Seed di numeri casuali
- **Se il bug non si verifica sempre, diventa ancora più complicato riuscire a capirne il motivo**



## 8) Divide et impera

- Individuare **le condizioni minimali** che rendono manifesto un bug
  - es. la stringa più breve, valore più piccolo
    - **Test dei casi limite è fondamentale**
    - Casi limite = Situazioni che possono portare il programma in errore

## 8) Divide et impera

- Individuare **le condizioni minimali** che rendono manifesto un bug
  - es. la stringa più breve, valore più piccolo
    - **Test dei casi limite è fondamentale**
    - Casi limite = Situazioni che possono portare il programma in errore
    - **Esempio:** calcolo del BMI
      - Valore limite: peso = 0
      - Il programma funziona con peso = 0 o da un errore? Se dà un errore, il bug è localizzato nel punto legato al calcolo del BMI
- **Le condizioni minimali possono facilitare la localizzazione di un bug**
  - Bisogna conoscere le condizioni minimali prima di cominciare a scrivere codice



## 9) Ricerca di regolarità

- **Alcuni bug si presentano con regolarità, ma non sempre**
- In questo caso, occorre capire il meccanismo ("pattern") che genera la regolarità



## 9) Ricerca di regolarità

- **Alcuni bug si presentano con regolarità, ma non sempre**
- In questo caso, occorre capire il meccanismo ("pattern") che genera la regolarità
  - Es: Alcuni bug si presentano solo dando in input numeri dispari
  - Es: Il bug si presenta solo dando in input valori negativi
  - **Es: Il bug si verifica solo se inserisco stringhe più lunghe di 10 caratteri**
    - Ecc. Ecc.
- Comprendere le regolarità **può aiutare a capire la natura del problema**
  - Es: Se il bug **si verifica solo se inserisco stringhe più lunghe di 10 caratteri** non memorizzo caratteri a sufficienza, e perdo delle informazioni.
    **Soluzione: Aumentare la dimensione del vettore per eliminare il bug.**



## 10) Stampe ausiliarie

- Per seguire l'esecuzione può essere utile **introdurre stampe ausiliarie**
  - Valido soprattutto per situazioni che non possono essere tracciate da un debugger es. sistemi distribuiti, programmi paralleli, etc. **Tipicamente si stampano i valori delle variabili**
  - **Adottare i meccanismi della ricerca binaria** ☺

## 10) Stampe ausiliarie

- Per seguire l'esecuzione può essere utile **introdurre stampe ausiliarie**
  - Valido soprattutto per situazioni che non possono essere tracciate da un debugger es. sistemi distribuiti, programmi paralleli, etc. **Tipicamente si stampano i valori delle variabili**
  - **Adottare i meccanismi della ricerca binaria** ☺
    - Es) Inserire una stampa a metà del programma. Se il valore è corretto, il bug è localizzato nella metà successiva. **Ripetere iterativamente il processo!**
- **Le stampe ausiliarie devono necessariamente essere eliminate** dopo aver scovato il bug
  - **Possono essere commentate anziché eliminate**
- Per situazioni complesse, si possono usare strumenti di logging
  - **Log =** Registrazione di tutte le operazioni effettuate dal programma



## Debugger

Debugger in Eclipse CDT 

- In alternativa (o in accoppiata) all'utilizzo di queste linee guida, si può (deve!) utilizzare **uno strumento chiamato debugger**
- **Un debugger guarda "dentro"** il programma durante l'esecuzione
  - **Tracing** del programma: *esecuzione istruzione per istruzione*
  - Visualizzazione del **contenuto delle variabili**
  - **Valutazione dinamica** di espressioni
  - **Breakpoint**, anche condizionali
  - **Stack trace:** sequenza di chiamate a funzione effettuate dal programma
  - …



## Debugger

Debugger in Eclipse CDT 

- **Un debugger guarda "dentro"** il programma durante l'esecuzione
  - **Tracing** del programma: *esecuzione istruzione per istruzione*
  - Visualizzazione del **contenuto delle variabili**
  - **Valutazione dinamica** di espressioni
  - **Breakpoint**, anche condizionali
  - **Stack trace:** sequenza di chiamate a funzione effettuate dal programma
  - …
- Sono strumenti molto sofisticati, **abituarsi al loro uso può migliorare significativamente la produttività nella programmazione.**



## Debugging in Eclipse CDT

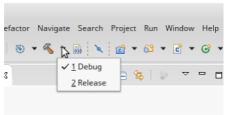
Debugger in Eclipse CDT 

- Un debugger **ha bisogno di informazioni aggiuntive** nel codice compilato
  - link tra il codice compilato e il codice sorgente
- Per stabilire la corrispondenza tra codice compilato e codice sorgente, **la compilazione per il debug non deve essere ottimizzata**

## Debugging in Eclipse CDT

Debugger in Eclipse CDT

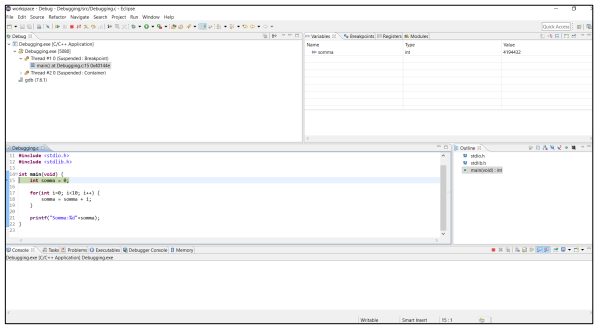
- Un debugger **ha bisogno di informazioni aggiuntive** nel codice compilato
  - link tra il codice compilato e il codice sorgente
- Per stabilire la corrispondenza tra codice compilato e codice sorgente, **la compilazione per il debug non deve essere ottimizzata**
- Due modalità di compilazione
  - **Debug**
    - **Meno efficiente, per il debug**
  - Release
    - Ottimizzata






## Debugging in Eclipse CDT

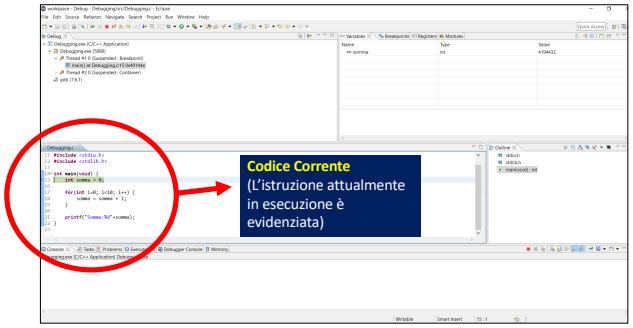
Debugger in Eclipse CDT





## Debugging in Eclipse CDT

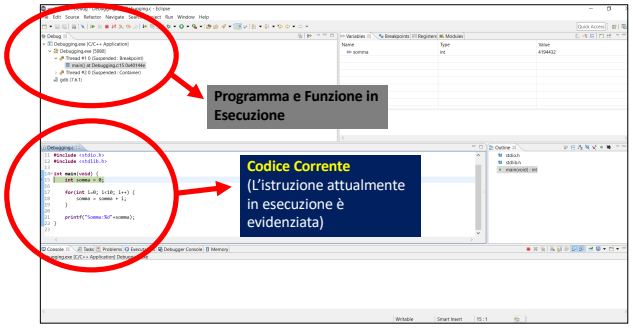
Debugger in Eclipse CDT





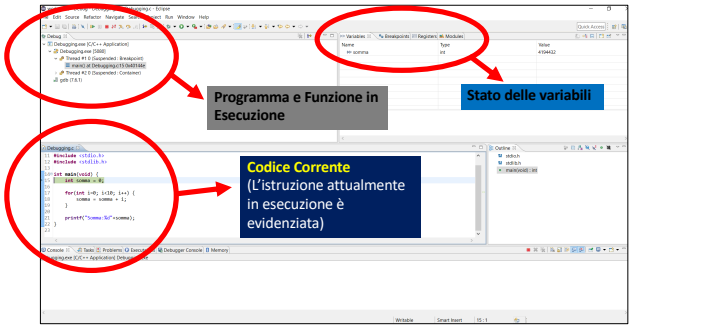

## Debugging in Eclipse CDT

Debugger in Eclipse CDT



## Debugging in Eclipse CDT

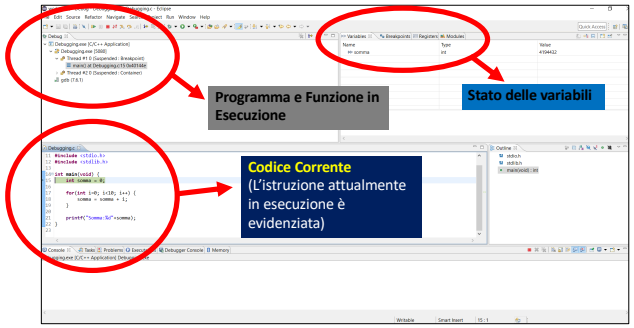
Debugger in Eclipse CDT






## Debugging in Eclipse CDT

Come controlliamo **l'esecuzione del programma?**





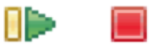

## Debugging in Eclipse CDT

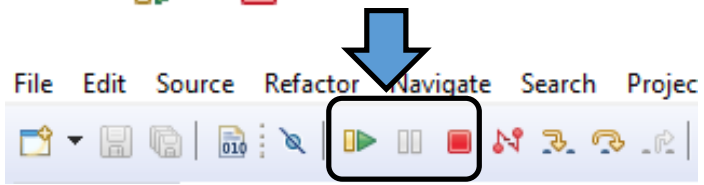
- Come controlliamo **l'esecuzione del programma?**



## Debugging in Eclipse CDT

- Come controlliamo **l'esecuzione del programma?**
  - **Comandi Resume e Terminate**





## Debugging in Eclipse CDT

- Come controlliamo **l'esecuzione del programma?**
  - **Comandi Resume e Terminate**

- **Resume**
  - Esegue le istruzioni fino al prossimo **breakpoint** oppure al **termine del programma**



## Debugging in Eclipse CDT

- Come controlliamo **l'esecuzione del programma?**
  - **Comandi Resume e Terminate**

- **Resume**
  - Esegue le istruzioni fino al prossimo **breakpoint** oppure al **termine del programma**
- **Terminate**
  - Interrompe l'esecuzione del programma
    - Utile quando il **programma va in loop infinito** o quando si scova un bug
  - **Attenzione:** i programmi non terminati rimangono in esecuzione per il sistema operativo
    - Occupazione inutile di memoria, Problemi per la ricompilazione



## Debugging in Eclipse CDT

**Problema:** per scovare più facilmente un bug abbiamo bisogno di eseguire il programma istruzione per istruzione

- Come controlliamo **l'esecuzione del programma?**
  - **Comandi Resume e Terminate**

- **Resume**
  - Esegue le istruzioni fino al prossimo breakpoint oppure al **termine del programma**
- **Terminate**
  - Interrompe l'esecuzione del programma
    - Utile quando il **programma va in loop infinito** o quando si scova un bug
  - **Attenzione:** i programmi non terminati rimangono in esecuzione per il sistema operativo
    - Occupazione inutile di memoria, Problemi per la ricompilazione



## Debugging in Eclipse CDT

- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:**
  - **Step over:**
  - **Step out:**


## Debugging in Eclipse CDT


- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche nelle** funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione



## Debugging in Eclipse CDT


- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche nelle** funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione

```
59        int i;
60        for (i=0; i<b; i++){
61            result = sum(result, a);
62        }
```

```
30  int sum(int a, int b) {
31      if (b == 0){
32          return a;
33      } else if (b > 0) {
```

**Step Into**



## Debugging in Eclipse CDT


- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche dentro** le funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione

```
59        int i;
60        for (i=0; i<b; i++){
61            result = sum(result, a);
62        }
```

```
59        int i;
60        for (i=0; i<b; i++){
61            result = sum(result, a);
62        }
```

**Step Over**



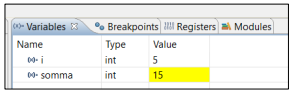
## Debugging in Eclipse CDT


- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche dentro** le funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione

- **Ogni click** sul tasto fa scorrere tra le istruzioni del programma.
  - In parallelo vengono aggiornati i valori delle variabili definite nel programma
  - **Box «Variables» in alto a destra**

## Debugging in Eclipse CDT

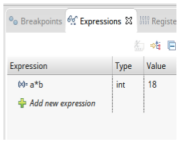

- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche dentro** le funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione
- **Ogni click** sul tasto fa scorrere tra le istruzioni del programma.
  - **Lo stato delle variabili viene aggiornato in tempo reale**
  - L'ultima variabile modificata **viene evidenziata**






## Debugging in Eclipse CDT

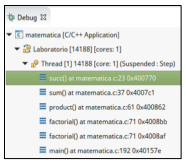

- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche dentro** le funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione
- **Ogni click** sul tasto fa scorrere tra le istruzioni del programma.
  - Durante l'esecuzione del programma può anche interessarci **valutare il valore a run time di alcune espressioni (es. espressioni di uscita dai cicli)**
  - **'Add new expression'**






## Debugging in Eclipse CDT


- Come controlliamo **l'esecuzione del programma?**
- **Tasti 'step'**
  - **Step into:** esegue il programma **entrando anche dentro** le funzioni
  - **Step over:** esegue il programma **ignorando** le funzioni
  - **Step out: torna alla funzione chiamante**, ignorando il contenuto della funzione
- **Ogni click** sul tasto fa scorrere tra le istruzioni del programma.
  - **Nel caso in cui vengano invocate delle funzioni, lo stack trace viene aggiornato**
  - **Box in alto a sinistra**






## Debugging in Eclipse CDT – Breakpoint

- (Finora) Come controlliamo **l'esecuzione del programma?**
  - **Tasti 'step'**
- Eseguire il programma istruzione per istruzione **può richiedere troppo tempo**

## Debugging in Eclipse CDT – Breakpoint

- (Finora) Come controlliamo **l'esecuzione del programma?**
  - **Tasti 'step'**
- Eseguire il programma istruzione per istruzione **può richiedere troppo tempo**
- **Alternativa: utilizzo dei breakpoint**



## Debugging in Eclipse CDT

- (Finora) Come controlliamo **l'esecuzione del programma?**
  - **Tasti 'step'**
- Eseguire il programma istruzione per istruzione **può richiedere troppo tempo**
- **Alternativa: utilizzo dei breakpoint**
  - Identificano dei punti del programma **che vogliamo 'monitorare'**
  - Si utilizzano **in corrispondenza di espressioni 'critiche'**



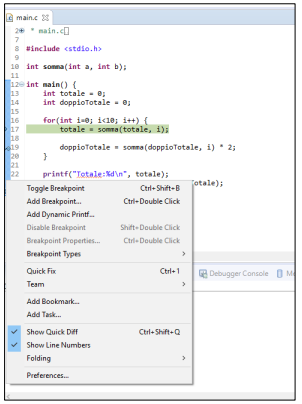
## Debugging in Eclipse CDT

- (Finora) Come controlliamo **l'esecuzione del programma?**
  - **Tasti 'step'**
- Eseguire il programma istruzione per istruzione **può richiedere troppo tempo**
- **Alternativa: utilizzo dei breakpoint**
  - Identificano dei punti del programma **che vogliamo 'monitorare'**
  - Si utilizzano **in corrispondenza di espressioni 'critiche'**
  - Il programma viene eseguito normalmente fino a quella istruzione, **poi il debugger si attiva e comincia a monitatore lo stato della macchina e delle variabili**
  - **Per definire un breakpoint, si effettua doppio click sul numero che identifica il rigo dell'istruzione (oppure tasto destro e 'Add Breakpoint')**



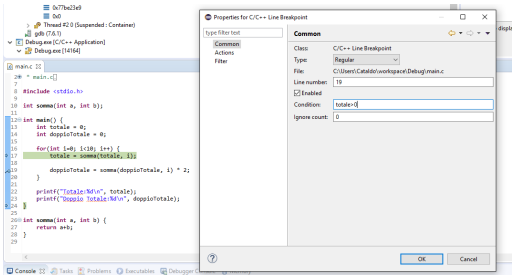
## Debugging in Eclipse CDT

- **Alternativa: utilizzo dei breakpoint**
  - **Per definire un breakpoint, si effettua doppio click sul numero che identifica il rigo dell'istruzione (oppure tasto destro e 'Add Breakpoint')**
  - Le istruzioni con un breakpoint vengono evidenziate accanto al numero di riga
  - E' possibile definire dei breakpoint più complessi?



## Debugging in Eclipse CDT

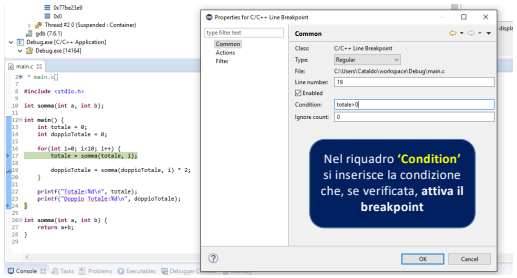
- **Breakpoint condizionali**
  - Particolare tipologia di breakpoint
  - L'esecuzione si ferma solo se viene verificata una particolare condizione
  - Come definirli?
    - Si definisce un breakpoint standard
    - Tasto destro → Breakpoint properties





## Debugging in Eclipse CDT

- **Breakpoint condizionali**
  - Particolare tipologia di breakpoint
  - L'esecuzione si ferma solo se viene verificata una particolare condizione
  - Come definirli?
    - Si definisce un breakpoint standard
    - Tasto destro → Breakpoint properties









## Debugger - Esercitazione

- Copiare nell'editor il codice sorgente mostrato nella prossima slide
- Il programma implementa un ciclo che ad ogni passaggio somma il valore dell'indice del ciclo alla somma. Il programma memorizza anche in una seconda variabile il doppio di questo valore.
- **Il programma ha un (semplice) bug.** Utilizzare il debugger per comprendere il bug presente
- Utilizzare il debugger, in tutte le sue funzionalità, per
  - Analizzare lo stack trace, cioè la sequenza delle funzioni chiamate
  - Analizzare il comportamento del debugger nelle funzioni **step into** e **step over**
  - **Seguire i valori delle variabili durante l'esecuzione per comprendere la natura dell'errore.**

# Debugger - Esercitazione

```
#include <stdio.h>
int somma(int a, int b); // prototipo di funzione

int main() {
        int totale = 0;  int doppioTotale = 0;

        for(int i=0; i<10; i++) {
                totale = somma(totale, i);
                doppioTotale = somma(doppioTotale, i) * 2;
        }

        printf("Totale:%d\n", totale);
        printf("Doppio Totale:%d\n", doppioTotale);
}

int somma(int a, int b) {
        return a+b;
}
```