# Parte 12

# Debugging

[Hieronymus Bosch – Hell, ~1500]

# Software bug

- http://en.wikipedia.org/wiki/Software_bug
- A **software bug** is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program or system that **produces an incorrect or unexpected result, or causes it to behave in unintended ways**
- **Bug reports**: reports detailing bugs in a program

# Bug effects (1)

- Some bugs have only a **subtle effect** on the program's functionality, and may thus lie **undetected** for a long time
- More serious bugs may cause the program to **crash** or **freeze** leading to a **denial of service**
- Others qualify as **security bugs** and might for example enable a malicious user to bypass **access controls** in order to obtain unauthorized privileges

# Bug effects (2)

- In 2002, a study commissioned by the **US Department of Commerce' National Institute of Standards and Technology** concluded that software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated **$59 billion annually, or about 0.6 percent of the gross domestic product**

# Crash

- A **crash** or (**system crash**) is a condition where a program (either an application or a part of the operating system) **stops performing its expected functions** and also **stops responding to other part of the system**
  - Possible causes: **illegal instructions** (divisions by zero, segmentation faults)
- If this program is a critical part of the operating system kernel, the entire computer may crash

# Freeze

- A **hang** or **freeze** occurs when either a single computer program or the whole system becomes unresponsive to inputs
  - E.g. Endless loops
- In a **freeze**, the window affected or the whole computer screen becomes static
- It differs from a crash, in which a program exits abnormally or the operating system shuts down
- When no other input works, the power cycle must be restarted by an **on/off or reset button**
- NOTE: a computer may seem to hang when it is simply processing very slowly

# Solutions

- Usually, in systems with a modern operating system, the user is able to terminate the programs running (for instance, with the **kill/pkill** command, or through the "**end task**" button on the task list in recent versions of Microsoft Windows)
  - and, if they wish, restart it in the hope that the anomalous condition that caused the hang does not recur.
- Older systems, such as those using MS-DOS or Windows 3.1x, often needed to be completely restarted in the event of a hang

# Common types of computer bugs (1)

## Arithmetic bugs

- Division by zero, Arithmetic overflow or underflow, Loss of arithmetic precision due to rounding

## Logic bugs

- Infinite loops and infinite recursion, off-by-one error (es. counting one too many or too few when looping)

## Syntax bugs

- Use of the wrong operator, such as performing assignment instead of equality test (in simple cases often warned by the compiler)

# Common types of computer bugs (2)

## Resource bugs

- Null pointer dereference, Using an uninitialized variable, Using an otherwise valid instruction on the wrong data type,  Resource leaks, Excessive recursion which though logically valid causes stack overflow

## Multi-threading programming bugs

- Deadlock, Race condition (collision between processes or threads on shared states)

# Debugging

- **Debugging is a methodical process of finding and reducing the number of bugs in a computer program**
  - **A very time consuming activity**: often programmers spend more time and effort finding and fixing bugs than writing new code; on some projects, more resources can be spent on testing than in developing the program
- Usually, the most difficult part of debugging is **finding the bug in the source code**; once it is found, correcting it is usually relatively easy

# The debugging process

- Typically, the first step in locating a bug is to reproduce it reliably
- Once the bug is reproduced, the programmer can use a **debugger or some other tool** to monitor the execution of the program in the faulty region, and find the point at which the program went astray
- However, some bugs may **disappear** when the program is run with a debugger!
  - "Heisenbug"

# Debuggers

- Programs known as **debuggers** exist to help programmers locate bugs by:
  - ***executing code line by line***
  - ***watching variable values***
  - ***other features to observe program behavior***
- **Without a debugger**, code can be added so that messages or values can be written to a console (for example with cout in the C++ language) or to a window or log file to trace program execution or show values

# A difficult task

However, **even with the aid of a debugger, locating bugs is something of an art**

- A bug in one section of a program may cause failures in a completely different section or in an apparently unrelated part of the system
- Sometimes, a bug is not an isolated flaw, but represents an error of thinking/planning (**logic errors**)
- Some classes of bugs have nothing to do with the code itself (e.g., related to hardware). It is common to change the code instead of the parts of the system, as the cost and time to change it is generally less

# Funzionamento del debugger

- Un debugger tipicamente prende in ingresso **soltanto il file eseguibile** da testare
  - Potenzialità limitate
  - E' necessario che l'eseguibile contenga informazioni che permettano di mettere in relazione ciascun dato o istruzione nell'eseguibile con la corrispondente entità nel sorgente (variabile, costante, istruzione, …)
- Se si dispone del codice sorgente da cui è stato ottenuto l'eseguibile il debugging può essere effettuato a **livello del sorgente stesso**

# Gdb

- In questa lezione faremo riferimento al debugger **gdb:** http://www.gnu.org/software/gdb/
- Per usare gdb su un eseguibile, è necessario che l'eseguibile sia stato compilato aggiungendo l'**opzione -g** (nel caso si usi gcc/g++)
- Tale opzione dice al compilatore di **memorizzare nel file eseguibile** una **tabella dei simboli** che include:
  - Nome dei simboli, tipo dei simboli, file e numeri di riga dei simboli

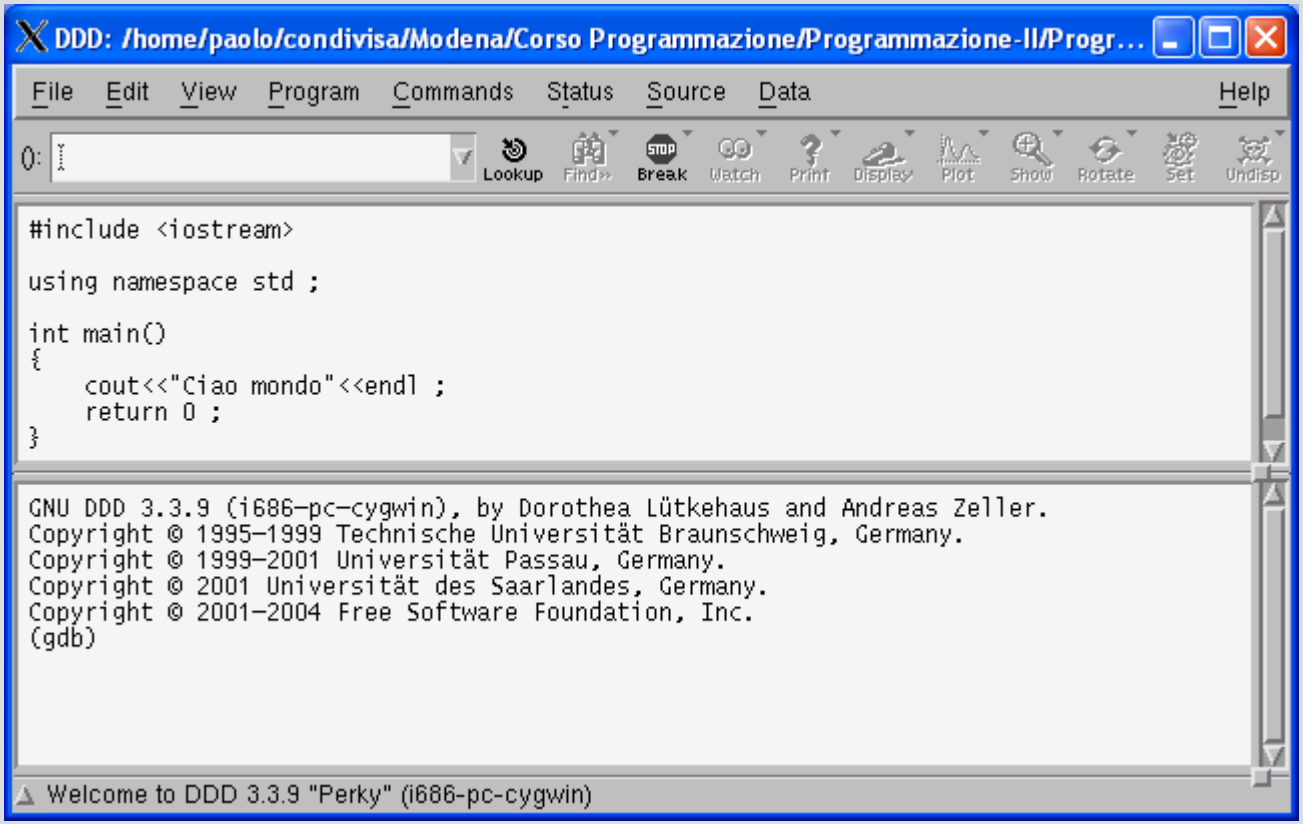
# ddd

- Non utilizzeremo direttamente **gdb**, ma il ***front-end grafico*** **ddd**
- http://www.gnu.org/software/ddd/
- Se si vuole lavorare da subito su un dato eseguibile si può **invocare ddd passando il nome dell'eseguibile**
- Ricordare che l'eseguibile deve essere stato compilato con l'opzione -g

# Proviamo

- Prendiamo il programma *ciao_mondo.cc*
- Compiliamolo opportunamente
- Invochiamo **ddd** passandogli l'eseguibile
- Se tutto è andato bene, **ddd** dovrebbe mostrarci il sorgente nella finestra in alto (**Source Window**), ed un terminale nella finestra in basso (**GDB Console**) con prompt **(gdb)**

# Finestra principale




**Source Window**

**GDB Console**

**Nella Console vediamo l'output e possiamo dare l'input al nostro programma (o eventualmente controllare direttamente gdb)**

# Command tool

- Inoltre dovrebbe esserci anche la finestra **Command Tool**
- Se non fosse visibile, selezionare View->Command Tool
- Permette di inviare **comandi a gdb**

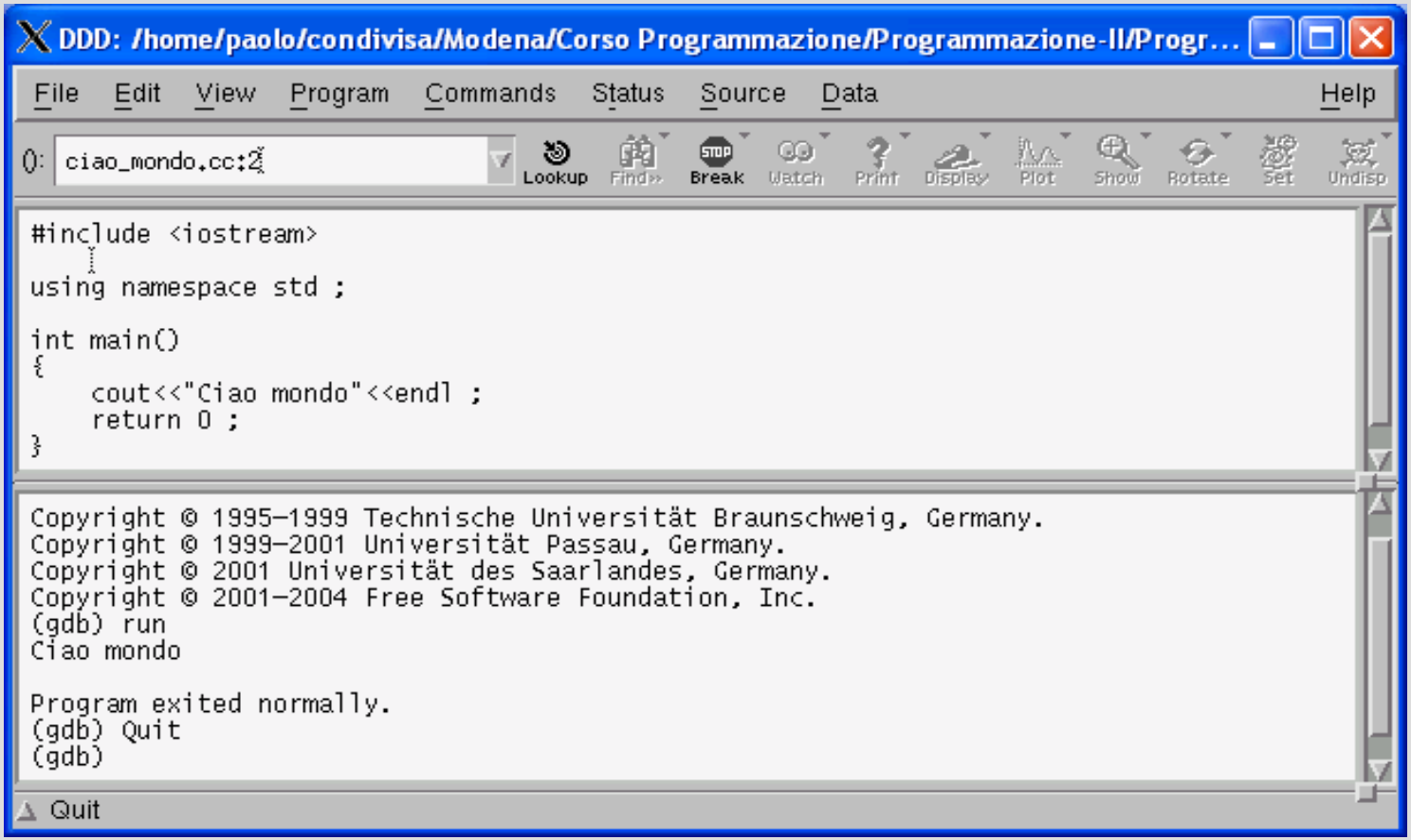
# Prima esecuzione (1)

- Proviamo subito ad eseguire il programma
  - Basta premere il tasto **Run**
- Se si fossero voluti passare argomenti al programma sarebbe stato necessario selezionare invece

  **Program->Run**
- **NOTA:** il comando **run,** seguito da eventuali parametri, può essere dato da **interfaccia grafica o da GDB console**
  - Scrivete **run** nella Console

# Prima esecuzione (2)

- Se tutto va bene, vediamo l'output del programma nella console



# Panoramica comandi

- Per farci una idea dei comandi disponibili leggiamo ora

  **man gdb**

- Fondamentale il concetto di **breakpoint *(comando break)*:**

  Quando il programma raggiunge un breakpoint si blocca, ed il controllo torna al debugger

# Uso dei breakpoint (1)

- L'importanza del debugger entra ovviamente in gioco quando si esegue il programma **passo-passo**
- Come facciamo a far fermare il programma prima della terminazione per poterlo eseguire passo-passo?
- Mettiamo un **breakpoint**

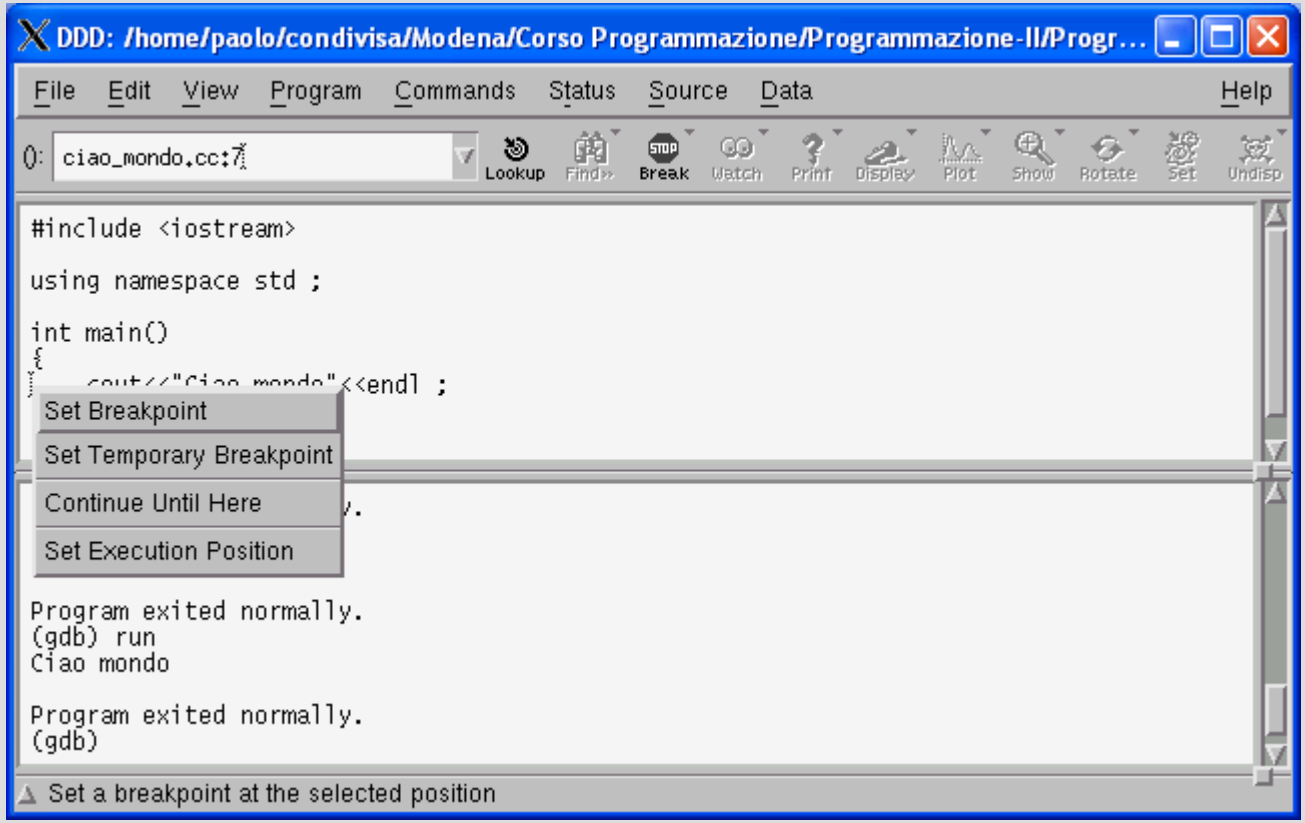
# Uso dei breakpoint (2)

Un modo per inserire un breakpoint è

- Puntare all'inizio della riga in cui vogliamo che il programma si fermi
- Premere il tasto destro del mouse
- Scegliere **Set Breakpoint**

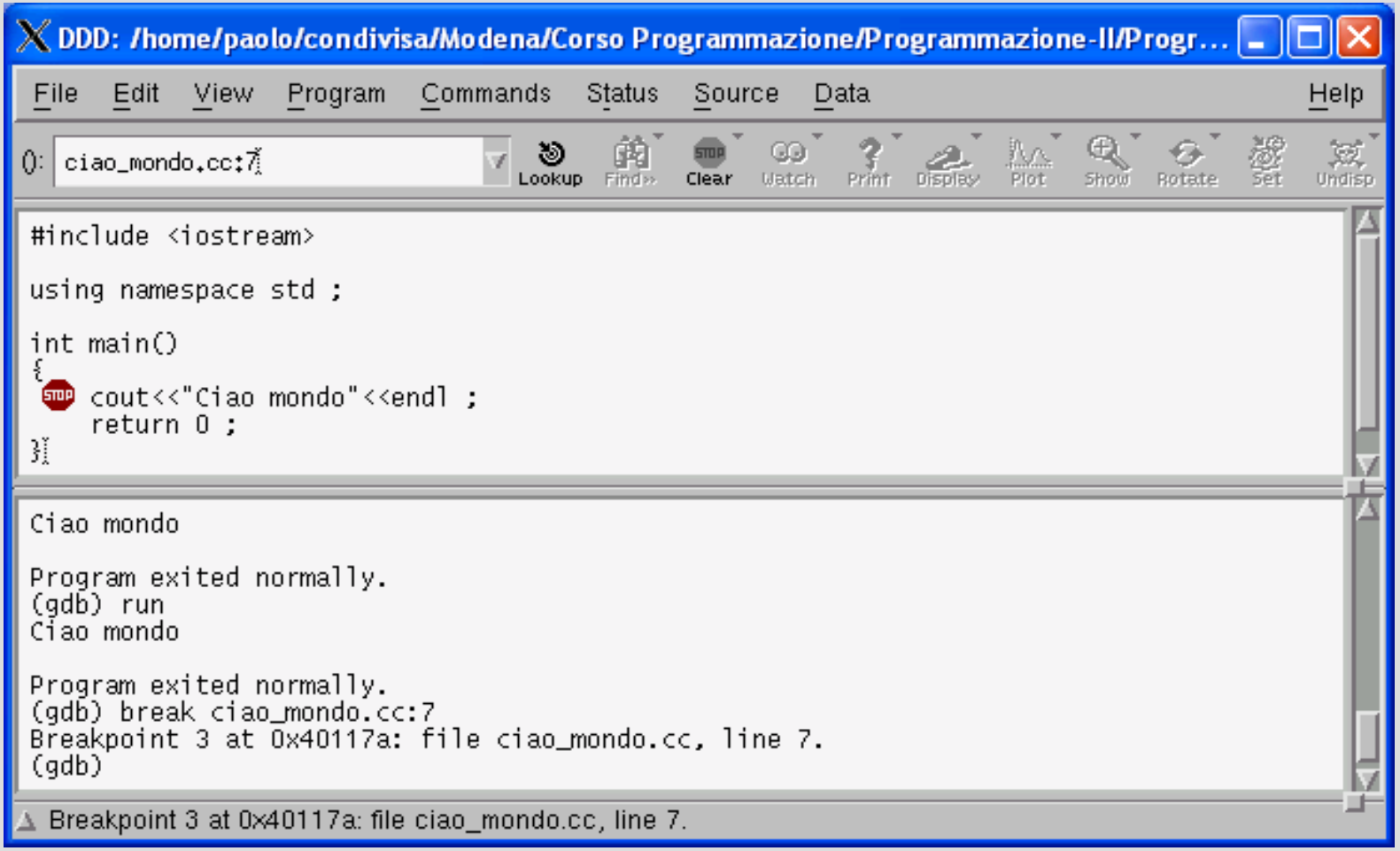
# Esempio

- Inseriamo un breakpoint sulla prima riga (riga di cout)



# Esempio

- Appare un'iconcina all'inizio della riga



# Stop su breakpoint

- Proviamo di nuovo ad eseguire il programma col comando **Run**
- Dovrebbe fermarsi sul **breakpoint**, ossia subito prima della stampa su **cout**
  - **Compare una freccia**
- Eseguiamo quindi il programma passo passo fino alla terminazione, mediante il comando (pulsante) **Next**
- Controlliamo che appaia **l'output**

# Codice esterno

- Durante l'esecuzione il programma può invocare **funzioni** o utilizzare **oggetti non definiti nel programma stesso**
- Come vedremo, questo può comportare l'**esecuzione implicita di parti di altri eseguibili**
- Se tale codice esterno non è stato compilato immettendo informazioni di debug, il debugger tipicamente segnala semplicemente il nome del file eseguibile attraversato

# Next e Step

- **Next e Step** eseguono entrambi il programma passo-passo, ma differiscono in quanto segue:
- **Next.** Esegue le chiamate di funzione senza 'entrarvi' dentro
- **Step.** Entra nel codice delle funzioni
  - Ovviamente il debugger non riesce ad entrare in funzioni definite in altri file eseguibili compilati senza includere informazioni di debugging

# Esempio

- Eseguire il programma *ciao_mondo2.cc* passo-passo
- Provare sia con **Next** che con **Step**
- ***Attenzione*** ad entrare con Step all'interno di funzioni non definite nel programma stesso: si rischia di dover eseguire molti passi prima di ritornare nel proprio programma!
- Proviamo un programma più complesso: es. *hs_sol.cc* (heapsort)
  - *Interagite col programma dalla Console*

# Ancora sui breakpoint

- Si possono mettere quanti **breakpoint** si vuole
- Con il comando **Continue**, si fa proseguire il programma:
  - fino al **prossimo breakpoint**
  - o **fino alla terminazione** se non vengono incontrati altri breakpoint

# Interruzione e terminazione

- Il comando **Interrupt** interrompe il programma e restituisce il controllo al debugger
- Il comando **Kill** termina definitivamente il programma

# Prossimo programma

- Nelle prossime slide faremo riferimento al programma *variabili_input_ciclo.cc*

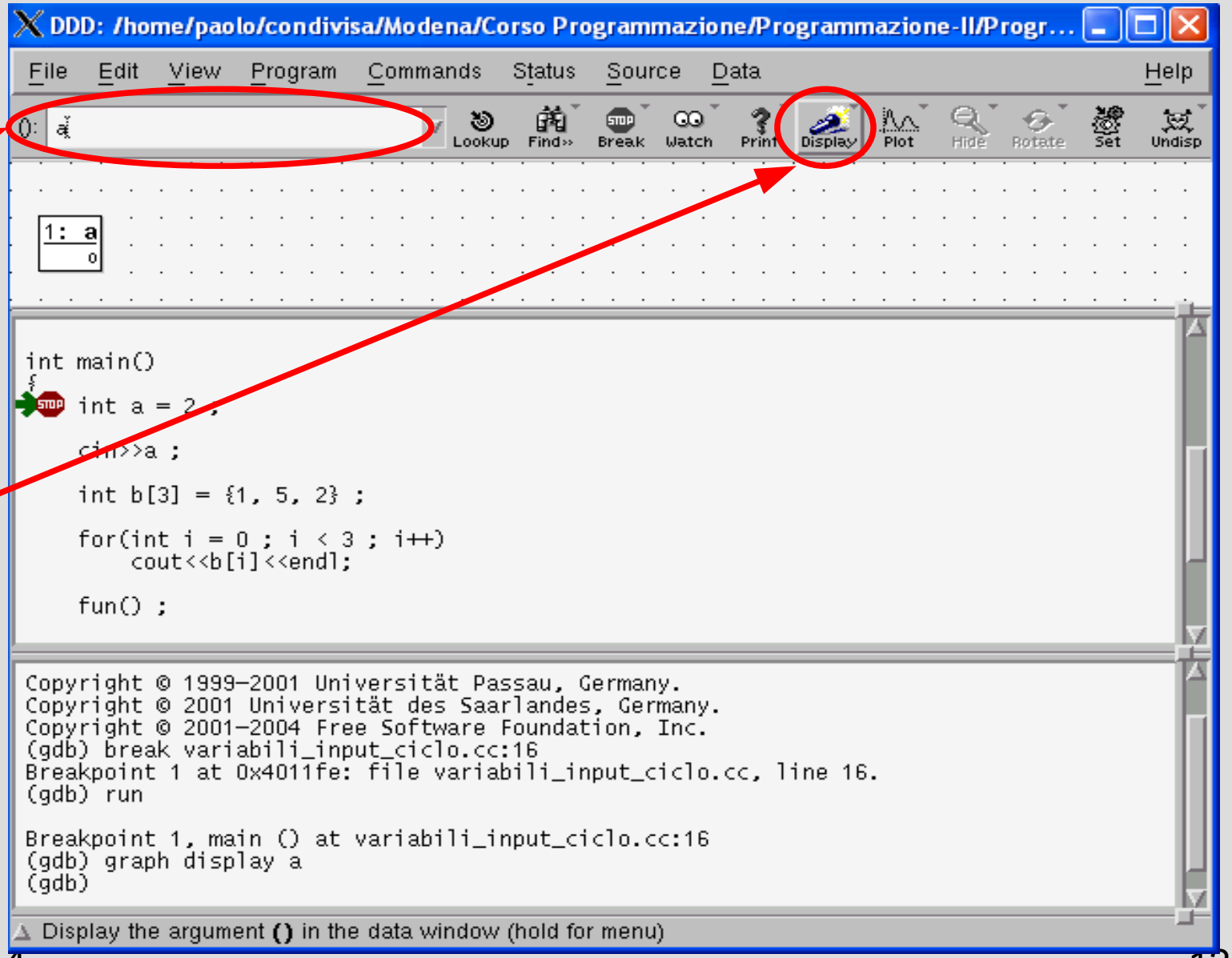
# Visualizzazione variabili (1)

- Il linguaggio C/C++ è un **linguaggio imperativo**
- Fondamentalmente l'esecuzione di un programma produce il **cambiamento di valore delle variabili**, ed è guidata dal valore delle variabili stesse
- Pertanto la **conoscenza del valore delle variabili** ci dice praticamente tutto sullo stato di un programma

# Visualizzazione variabili (2)

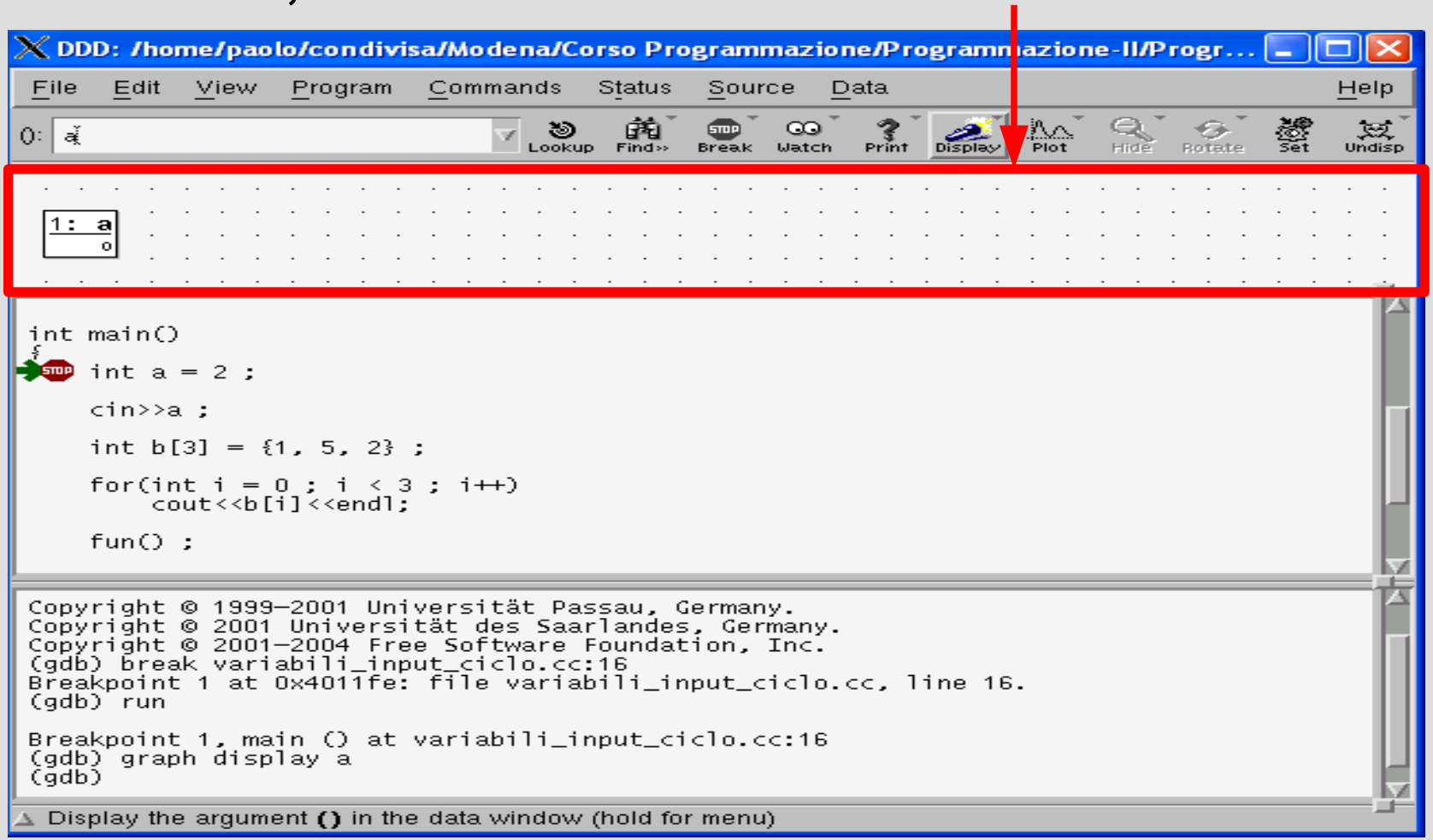
- Una delle funzioni di un debugger è proprio la **visualizzazione delle variabili**
  - In realtà un debugger può anche **forzare** il valore delle variabili, ma non approfondiremo questo aspetto
- Vi sono **vari modi** di visualizzare il valore di una variabile con **ddd**
  - Uno è scriverne il nome nel campo argomenti e premere il tasto **Display**, come illustrato nella prossima figura
  - *NOTA: bisogna essere in stato di run sospeso*

# Visualizzazione variabili (3)



# Visualizzazione variabili (4)

- Come si vede, si è aperta una ulteriore finestra, chiamata **Data Window**



# Visualizzazione variabili (5)

- Un modo ancora più veloce per far apparire il valore di una variabile nella **Data Window** è farvi **doppio click** sopra
- La scatolina all'interno della quale è mostrata la variabile nella **Data Window** è chiamata **Display**
- Per eliminare un **Display** si può premere il tasto destro del mouse sul nome della variabile nel **Display** e selezionare **Undisplay**

# Visualizzazione veloce

- Se si sposta semplicemente il puntatore su una variabile appare un piccolo riquadro con il valore corrente della variabile

# Visibilità e tempo di vita

Affinché il valore di una variabile sia correttamente visualizzabile:

- il programma deve essere **in esecuzione e sospeso**
  - **Run con breakpoint**
- la variabile deve essere **allocata in memoria e visibile** nel punto in cui il programma è fermo
  - Se entriamo nella funzione fun, la variabile a del main scompare dalla **Data Window**

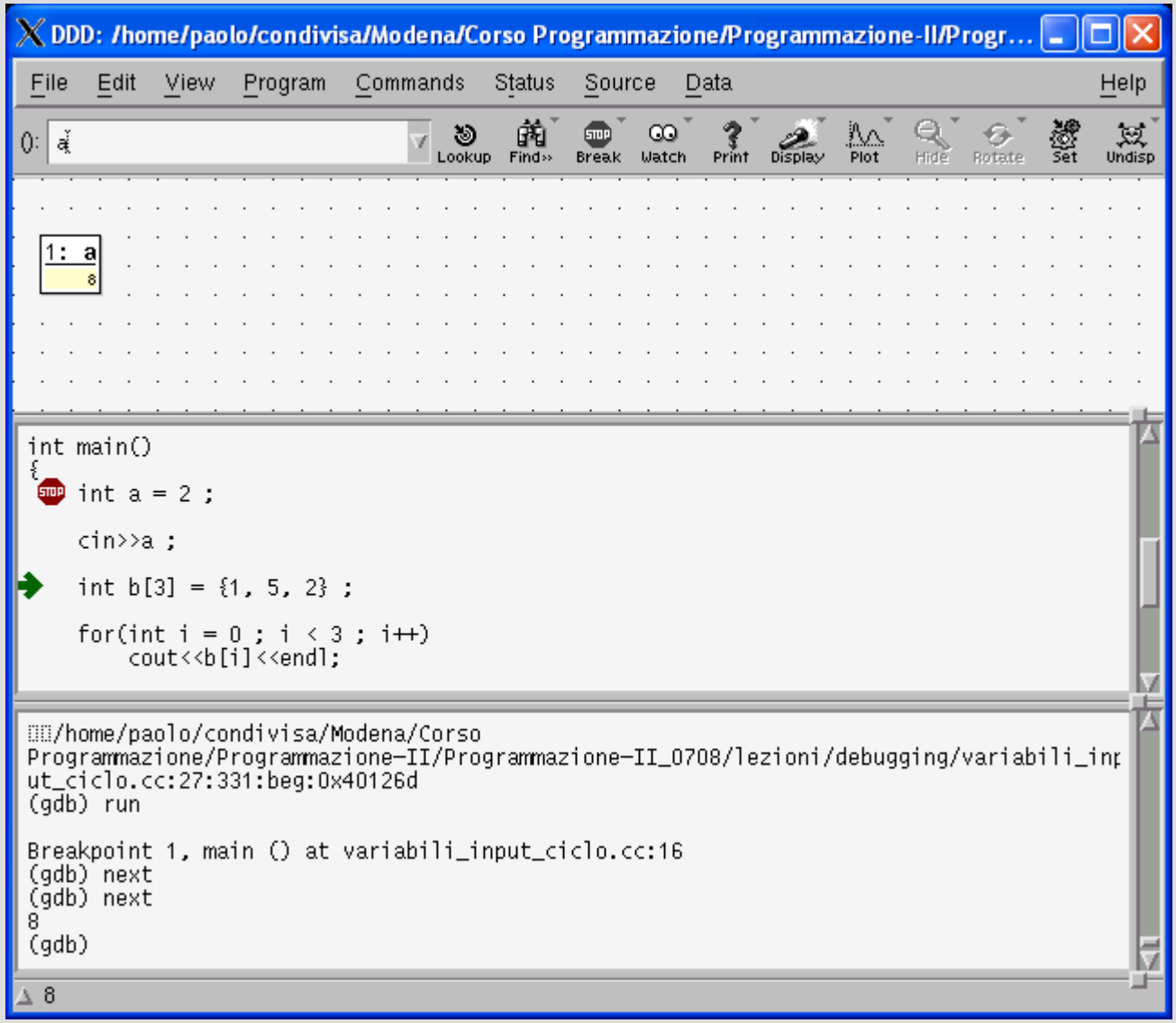
# Continuiamo con l'esempio

- Eravamo fermi sulla prima istruzione con la variabile **a** visualizzata nella **Data Window**
- Mediante **Next** eseguiamo il programma passo-passo
- Vediamo che appena iniziamo l'esecuzione dell'operatore di ingresso su **cin** il programma si blocca e non va più avanti

# Esempio

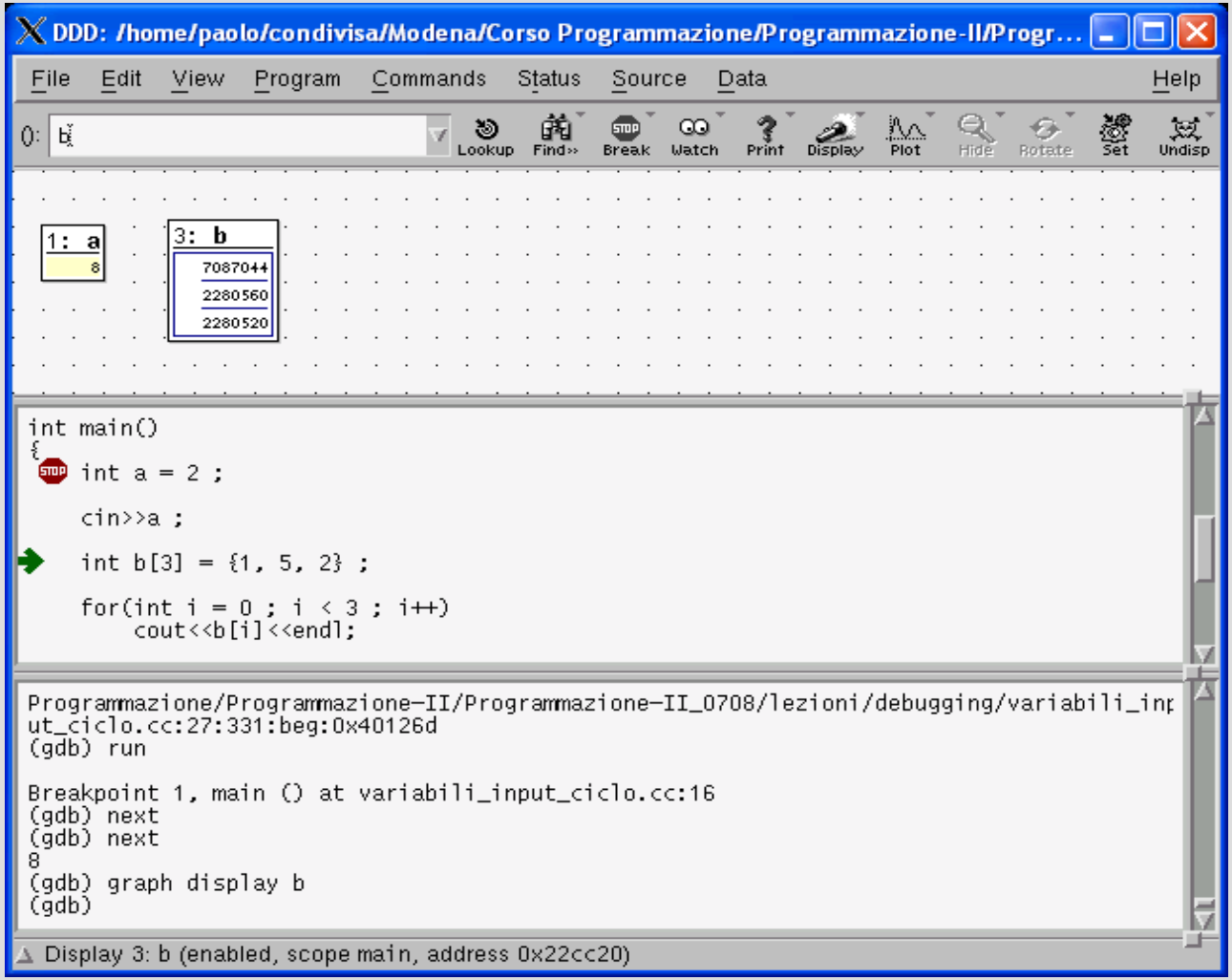
- Il programma sta aspettando che immettiamo un valore intero da **stdin**
- Quindi clicchiamo sulla **GDB Console** ed immettiamo il valore
- Se tutto è andato bene dovremmo vedere il nuovo valore della variabile nella **Data Window**, mentre la prossima istruzione da eseguire dovrebbe essere quella successiva all'operatore di ingresso, come in figura seguente

# Nuovo stato


DDD: /home/paolo/condivisa/Modena/Corso Programmazione/Programmazione-II/Progr...
File Edit View Program Commands Status Source Data Help
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
1: a
8
int main()
{
int a = 2 ;
cin>>a ;
int b[3] = {1, 5, 2} ;
for(int i = 0 ; i < 3 ; i++)
cout<<b[i]<<endl;
/home/paolo/condivisa/Modena/Corso Programmazione/Programmazione-II/Programmazione-II_0708/lezioni/debugging/variabili_input_ciclo.cc:27:331:beg:0x40126d
(gdb) run
Breakpoint 1, main () at variabili_input_ciclo.cc:16
(gdb) next
(gdb) next
8
(gdb)
8

# Visualizzazione array

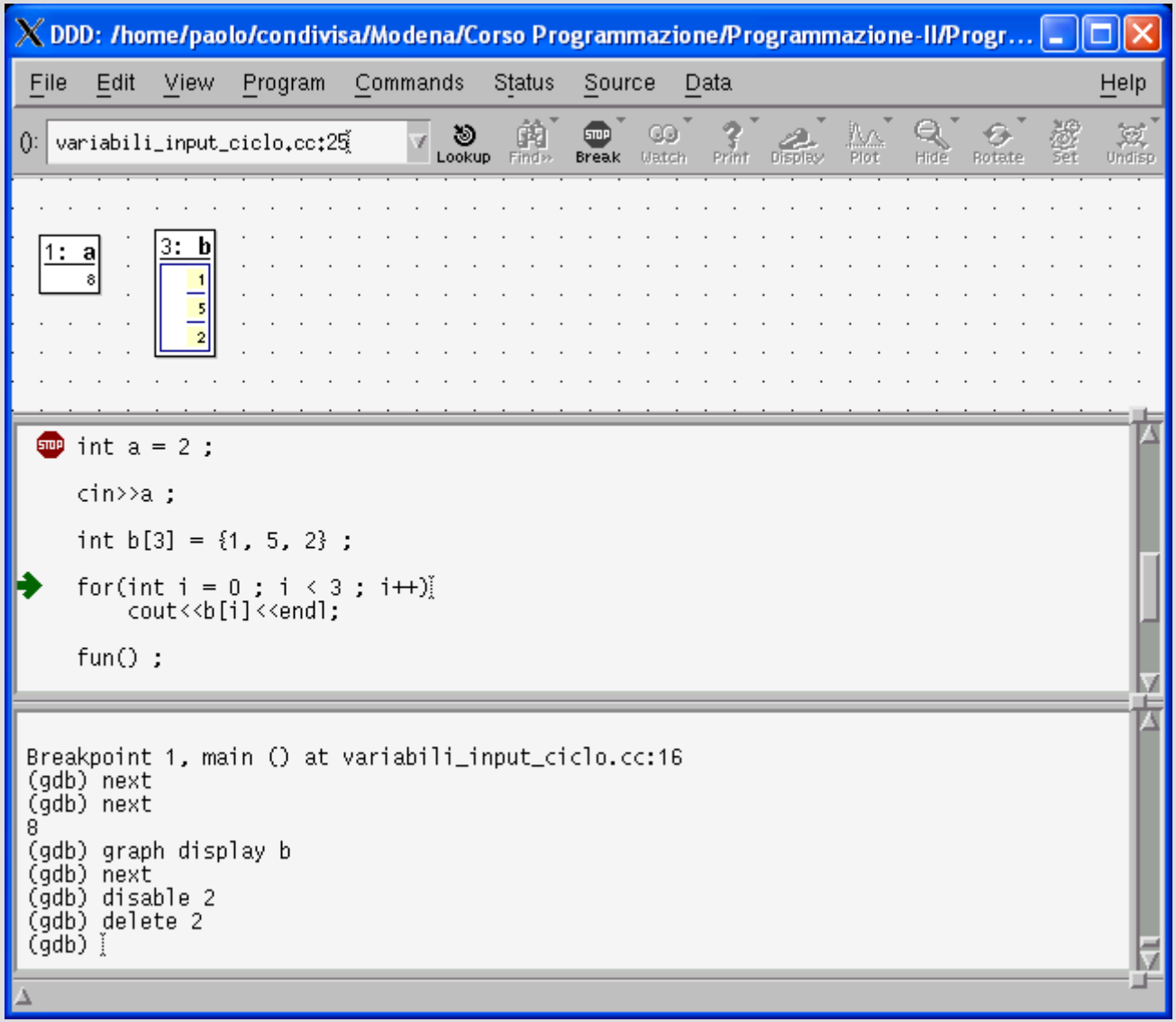
- Fare doppio click sulla variabile b



# Valori casuali

- Appena visualizzato, gli elementi dell'array hanno valori **casuali**
  - Perché siamo prima dell'inizializzazione!
- Premiamo quindi **Next** per eseguire l'inizializzazione

# Dopo l'inizializzazione

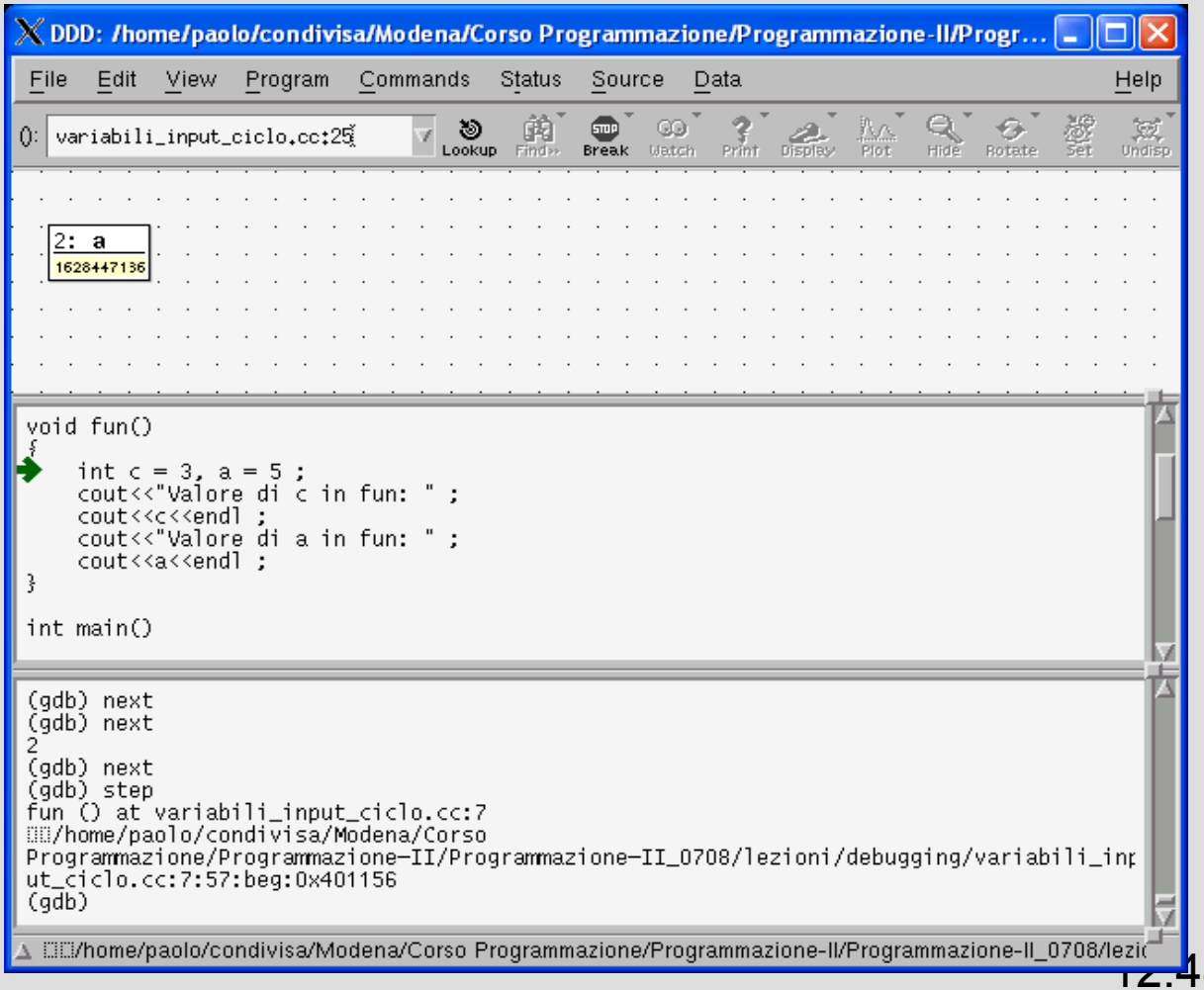


# Esecuzione ciclo  e funzione

- Premendo più volte il tasto **Next** si può osservare l'esecuzione del ciclo for per la stampa dei valori degli elementi dell'array
- Fermarsi subito prima dell'invocazione della funzione **fun**
- Premendo **Step** entrare nella funzione **fun**

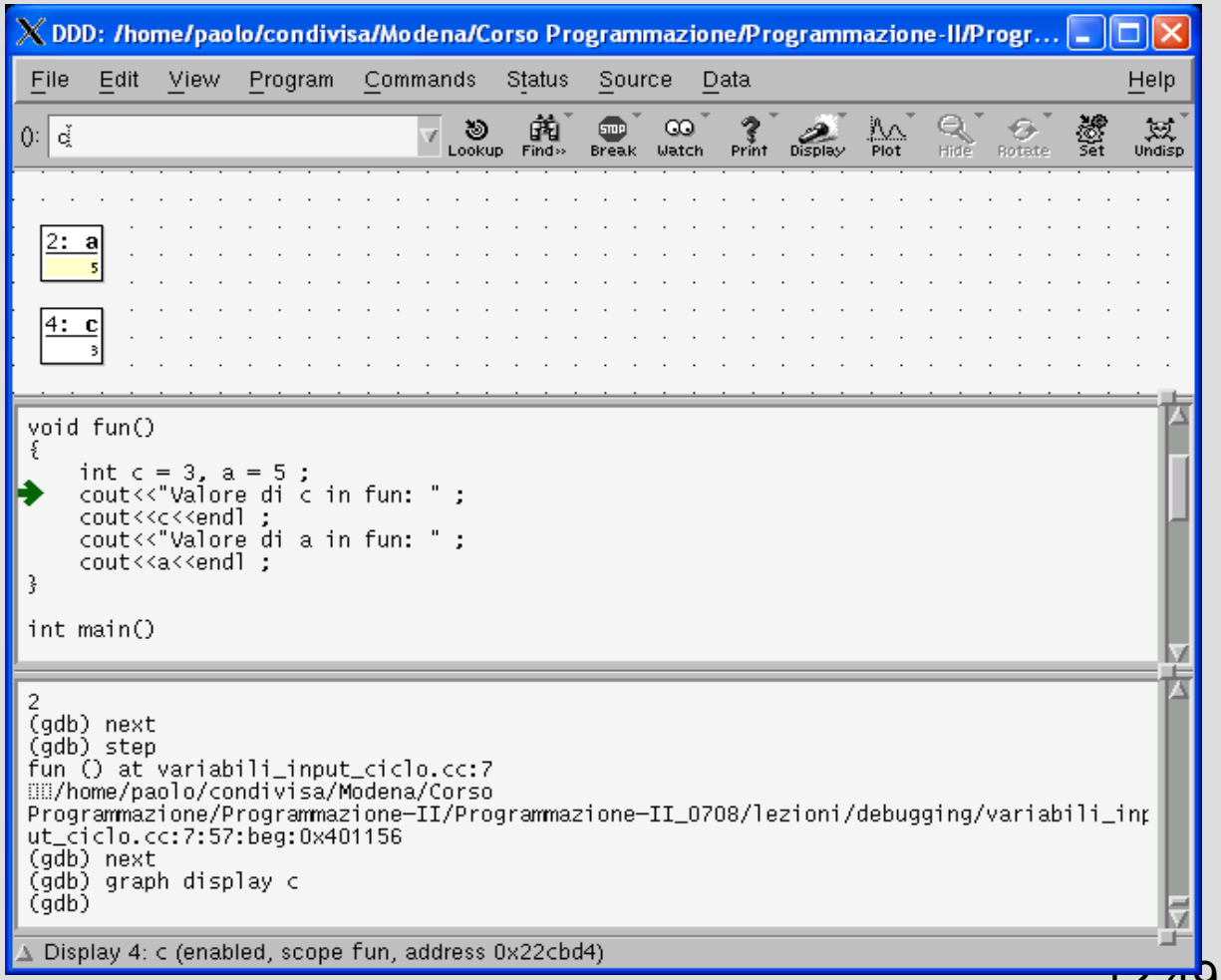
# Entrata in una funzione

- Come si vede, il display della variabile e dell'array sono spariti
- *Visibilità delle variabili*
- *Visualizziamo la variabile a locale alla funzione fun*



# Prosecuzione

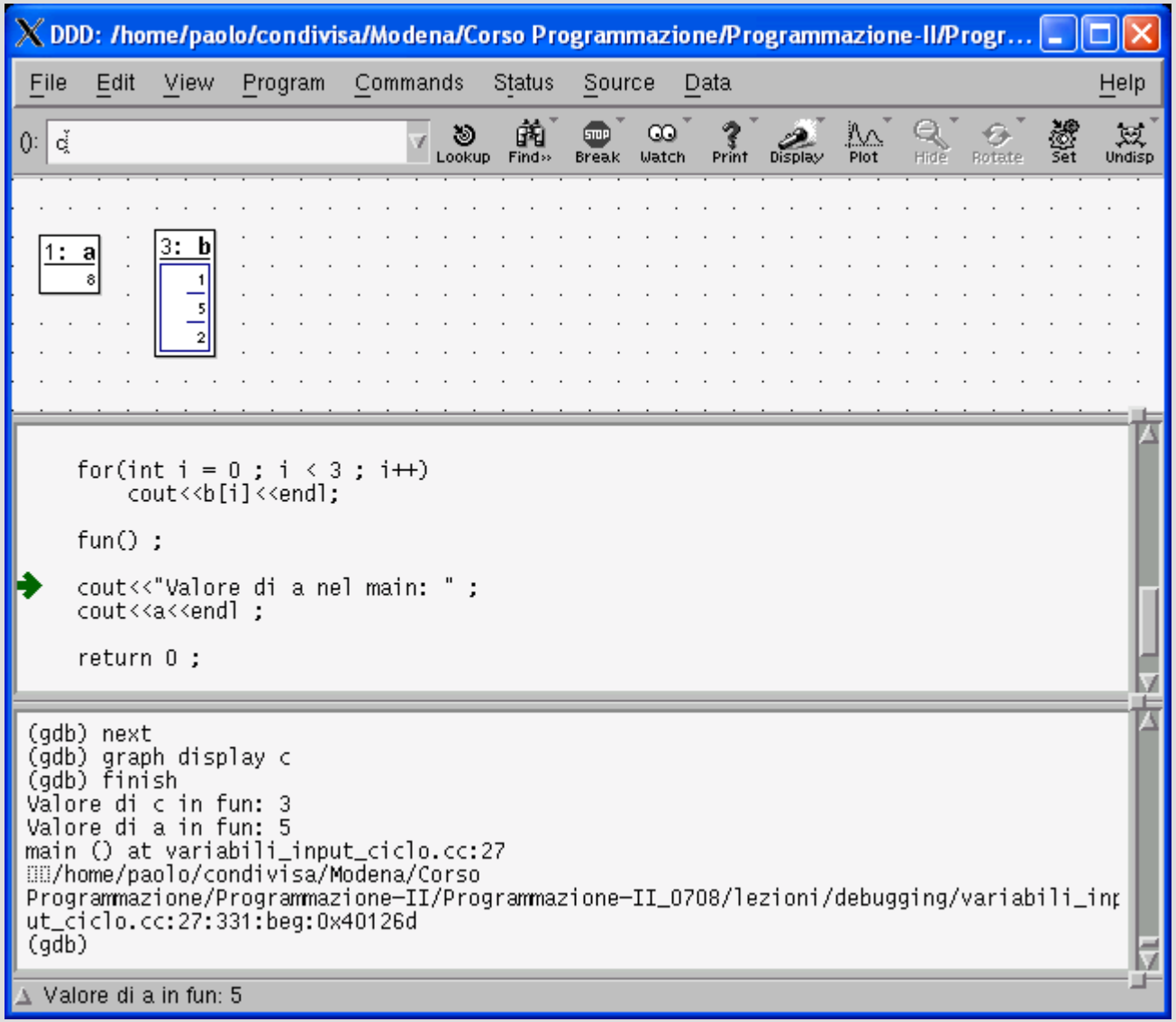
- Facciamo apparire anche il valore della variabile **c** nella **Data Window**



# Finish

- Si può far completare l'esecuzione della **funzione corrente** premendo **Finish**
  - **Kill** provoca invece la **terminazione definitiva di tutto il programma**
- Il programma si sospende subito **dopo la terminazione della funzione** ed il controllo torna al debugger

# Nuova situazione




Come si può notare, sono ricomparsi i **Display** di **a** e **b**, perché le due variabili sono tornate visibili

# Variabili e parametri di una funzione

- Si possono visualizzare tutte le variabili locali di una funzione con un solo comando:

  **Data->Display Local Variables**

- Si possono visualizzare tutti i parametri formali di una funzione con un solo comando:

  **Data->Display Arguments**

- *Provate...*

# Selezionare i Display

- Si possono selezionare più **Display** contemporaneamente, per esempio tenendo premuto il tasto **shift** e cliccando su tutti i **Display** da selezionare
  - Operazioni successive, quali per esempio l'eliminazione o gli spostamenti riguarderanno tutti i **Display** selezionati
- Non vedremo ulteriori dettagli su **ddd**... fare riferimento alla documentazione

# Esercizio

Utilizzando **solo il debugger** trovare eventuali errori nei programmi

- *residuo.cc*
- *tabella_ASCII_inversa.cc*
  - Dove mettiamo il breakpoint?
  - Bisogna trovare il modo di fermare il programma solo dopo un certo numero di iterazioni

*Soluzioni:*

1. Mettiamo un breakpoint su: if ( i % 8 == 0 )
2. Forziamo il valore della variabile i

# Visualizzazione array

- Se l'**array è statico**, lo si visualizza semplicemente scrivendo il suo nome nel Campo Argomenti e cliccando su Display
- Se l'**array è dinamico**, questo metodo non funziona:
  - Visualizza solo il puntatore all'array
  - *Problema*: ddd non conosce la lunghezza dell'array
- Esempio: *array_dinamici.cc*

# Visualizzazione array dinamici

- Attenzione al comando **gdb** che compare nella **GDB Console** se proviamo a visualizzare l'array A cliccandoci sopra:

  *graph display A*
- Proviamo con:

  *graph display A[0]*

  *graph display A[0]@num_elem*
- **NOTA**: *num_elem* può essere sostituito da un semplice intero che indica il numero degli elementi da visualizzare

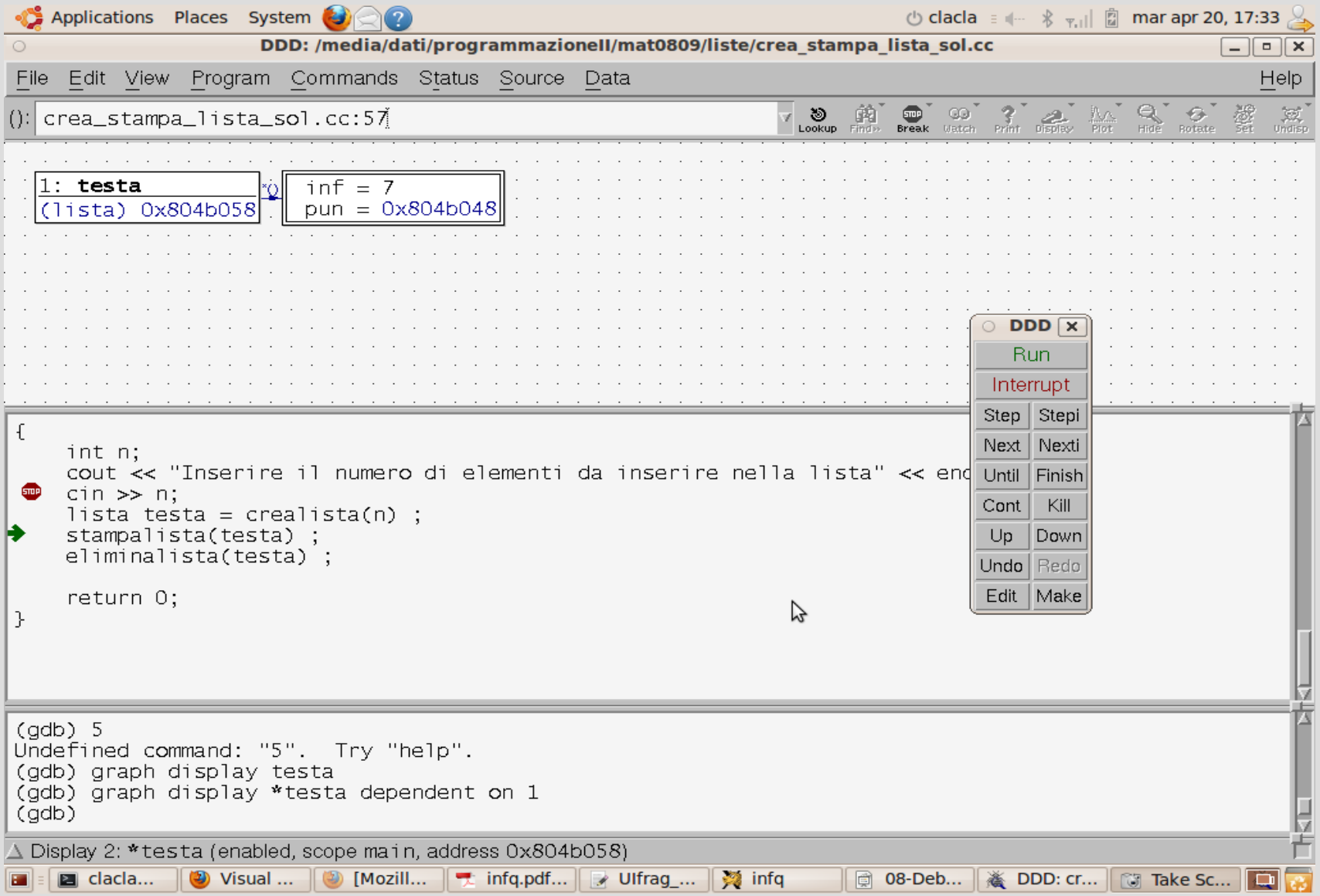
# Visualizzazione struct

- **ddd** permette di visualizzare agevolmente una struttura dati composta, quale una **lista composta da elementi concatenati** (*struct*)

  *crea_stampa_elim_lista_sol.cc*

- Proviamo a visualizzare la lista dopo averla creata
  - Cliccando su **testa** – puntatore alla testa della lista – si visualizza solo il valore del puntatore
  - Inseriamo nel **Campo Argormenti** la stringa “***testa**” e clicchiamo su **Display**

# Visualizzazione primo elemento

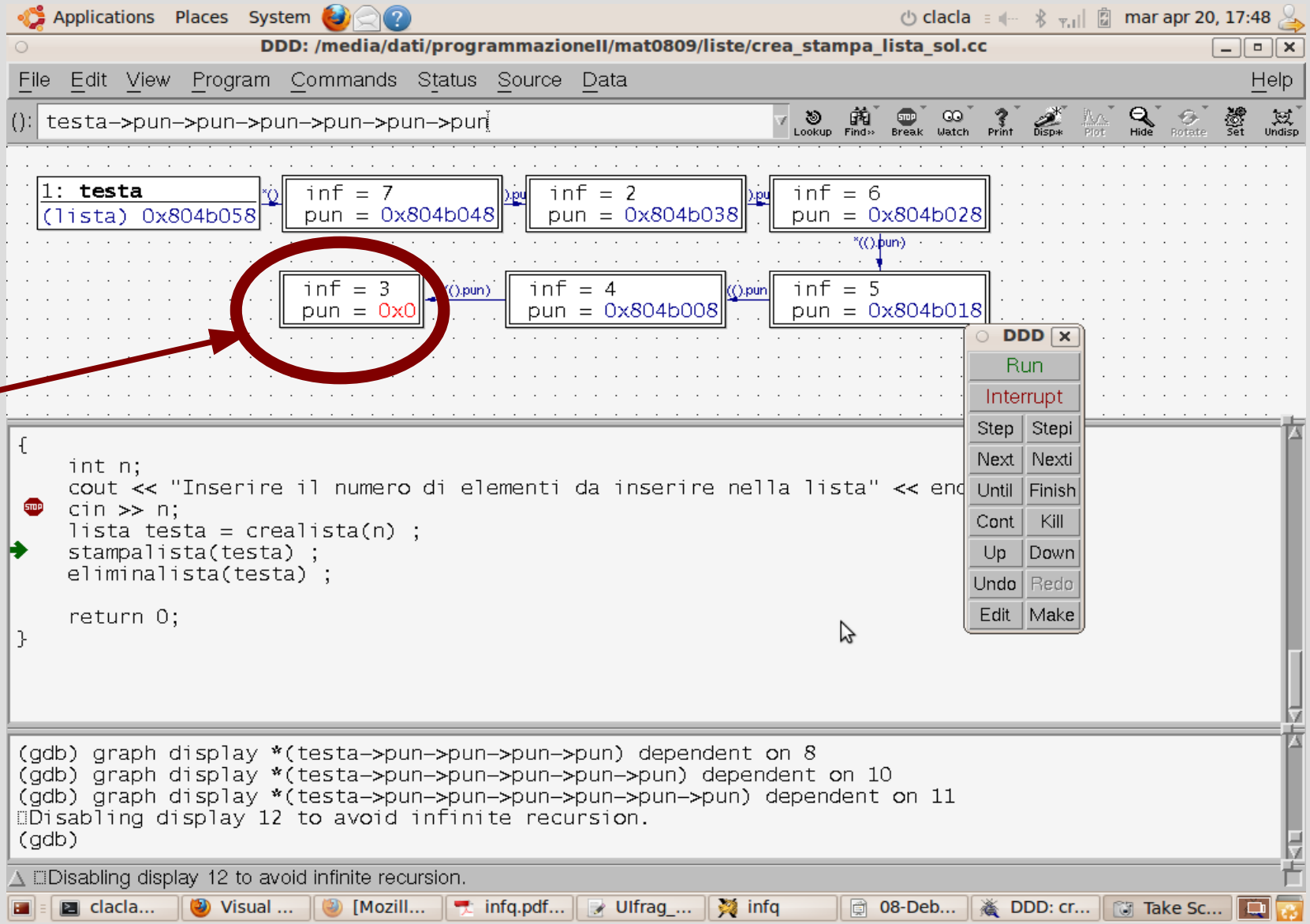


# Visualizzazione elementi successivi

- All'elemento successivo al primo, si accede attraverso il puntatore **pun**
- Per visualizzarlo, facciamo doppio click su **pun**
- Il secondo elemento appare in coda
- Notare i *collegamenti* attraverso gli elementi della lista
  - Potete spostare per trascinamento gli elementi nella Data Window
  - I collegamenti rimangono

# Visualizzazione elementi successivi

**NOTA**:
Il campo **pun** dell'ultimo elemento è a 0!



# Osservazione

- La **visualizzazione grafica** di una lista di elementi con i relativi collegamenti può essere molto utile in fase di debugging
- Ricordate la difficoltà delle fasi di testing e debugging dei programmi sulle liste!
  - L'utilizzo di stampe in questo caso è molto limitativo (oltre che scomodo)

# Torniamo ai bug...

## Unusual bugs

- Classe di software bugs considerati **eccezionalmente difficili da scovare e/o correggere**
- Ne esistono di diversi tipi
  - *Heisenbug*
  - *Mandelbug*
  - *Statistical bug*

# Heisenbug

Un **Heisenbug** è un bug che scompare o altera le sue caratteristiche quando si tenta di identificarlo

- Un esempio comune è un bug che si riscontra in un programma compilato con un **optimizing compiler**, ma non nello stesso programma compilato senza ottimizzazione (es. creando una versione per il debug)

# Compilatore ottimizzato

- **Compiler optimization** è il processo di tuning dell'output di un compilatore per utilizzare meno risorse
  - Attuato attraverso sequenze di trasformazioni e algoritmi
- Requisito più comune: **minimizzare il tempo di esecuzione** di un programma
- Altro requisito: **minimizzare la quantità di memoria occupata**

# Tracing

# Tracing

- Esiste un'altra possibilità per effettuare il debugging oltre all'uso del debugger: il **tracing**
- Nell'ingegneria del software, il **tracing** è un **uso specializzato del logging** per memorizzare informazioni circa l'esecuzione di un programma
- Tipicamente usato dai programmatori per il debugging o dal supporto tecnico per diagnosticare problemi col software

# Opinioni

- *“The most effective debugging tool is still careful thought, coupled with judiciously placed print statements”*
- Brian Kernighan, "Unix for Beginners" (1979)

# Istruzioni di stampa

- Aggiungere semplicemente delle stampe comporta **due problemi principali:**
  1. **Peggioramento leggibilità**
  2. **Difficoltà di eliminazione**

# Peggioramento della leggibilità

Le istruzioni di stampa a scopo di tracing/debugging si confondono col resto del codice (in particolare con le altre istruzioni di stampa)

# Difficoltà di eliminazione

- Una volta che si è trovato l'errore, bisogna **eliminare le istruzioni di stampa** nella versione definitiva del programma
- Molto spesso però poi capita di doverle **riusare** quando si presenta un nuovo bug
- Allora è meglio magari **commentarle** per poterle riattivare quando necessario
  - **Ma la presenza di istruzioni commentate peggiora ulteriormente l'aspetto del codice**
- **Aspetto critico:** per commentare/eliminare, occorre agire su **tutte** le istruzioni **una ad una!**

# Macro

- Una soluzione molto più pratica è offerta dalle **macro**
- Definite attraverso le **direttiva per il preprocessore**

  **#define**

  permettono **sostituzioni testuali** prima dell'inizio della compilazione stessa

# Tracing macros

- Possiamo ora vedere come usare le **macro per il tracing**
- Potremmo aggiungere all'inizio del programma

  **#define DEB(a)  cout<<(a)<<endl ;**
- Oppure, in modo ancora più generale

  **#define DEB(a)  {a ;}**
- Facendo in questo secondo modo possiamo passare come argomento anche frammenti di codice arbitrari

# Esempio

- // calcolo di somma e massimo di 5 interi

```
main()
{   int numero, max, somma ;
    for (int i=1; i <= 5; i++) {
          cin>>numero ;
          somma += numero;
          if(numero>max)
                max = numero;
     }
cout<<"Max: "<<max<<",somma: "<<somma<<endl ;
}                                   // il programma non funziona !
```

- Effettuiamo il tracing con le macro: *somma_max_tracing.cc*

# Leggibilità

- Vedendo **DEB**, il lettore del codice capisce chiaramente che quella istruzione è relativa al **debugging/tracing**

# Esercizio

- Per rendere il tracing **meno verboso** (prolisso), eliminare la stampa dei valori di somma e max subito dopo la **cin**
- Devono restare solo le stampe effettuate dopo aver aggiornato tali valori
- *Programma somma_max_meno_verb.cc*

# Commento

- Si è aggiunta una **seconda macro vuota**
- Questa è una soluzione pratica, perché per disabilitare un certo messaggio basta sostituire **DEB** con **DDB** per quel messaggio
  - Gli editor di testo consentono di effettuare l'operazione in modo **veloce e pratico**
- La **riattivazione** è altrettanto veloce

# Livelli di debug (1)

- Con i trucchi di sostituzione mediante editor siamo ancora ad un livello un po' **artigianale**
- Si può avere una soluzione ancora più pulita ed efficiente
- Si può prevedere che esistano **diversi livelli di debug**, tali che, per ciascun livello si abbiano messaggi di tracing più o meno verbosi

# Livelli di debug (2)

- Per implementare i **livelli di debug,** possiamo utilizzare le **macro** con **due argomenti** e le **maschere binarie**
- Vediamo un semplice esempio in *somma_max_tracing_liv.cc*
- E' abilitato solo un livello di debug: **D1**

# Domanda

1. Che valore bisogna dare alla **maschera** per abilitare tutti e due i livelli?
2. E per abilitare solo l'altro livello?

# Risposta

1. Il valore 1|2 oppure direttamente 3
2. Il valore 2

# Domanda

- Come mai la **maschera** è stata definita come una variabile e non come una costante?

# Maschera variabile (1)

- Definendo la **maschera** come una variabile abbiamo l'ulteriore libertà di poterne cambiare il valore durante l'esecuzione del programma
- Questo può servire, per esempio, quando un programma fallisce dopo avere eseguito un gran numero di istruzioni

# Maschera variabile (2)

- In questi casi possiamo mettere una **istruzione condizionale** nel programma che modifichi la maschera riattivando i livelli di debug desiderati solo quando si verifica una condizione sperabilmente vicina al punto in cui il programma fallisce
- Possiamo così evitare una tempesta di messaggi iniziali che possono rallentare tantissimo il programma e rendere l'interpretazione dei messaggi stessi molto più impegnativa

# Esempio

- Supponendo che **mask** sia inizializzato a 0, se pensiamo che il seguente ciclo fallisca dopo 10000 iterazioni, possiamo evitare i primi 10000 messaggi così:

```
for(int i = 0 ; i < 20000 ; i++) {
        if (i == 10000)
        mask |= 1 ;
        ...
        DEB(1, cout<<...messaggio...<<endl) ;
}
```

# Modalità di debug (1)

Spesso nella programmazione professionale conviene poter compilare un programma in almeno due **modalità distinte**

- Una **modalità di debug e/o testing** in cui sono attivi tutti i controlli ed i messaggi di tracing che si ritengono necessari
- Una **modalità 'normale'** in cui si è disabilitato tutto il codice di debugging

# Modalità di debug (2)

- La **modalità di debug** torna ovviamente comoda in fase di **sviluppo** del programma
- Quella **normale** si può utilizzare poi quando il programma viene veramente utilizzato per il suo **scopo finale** o quando è **consegnato al committente**
- Le **macro** ci consentono di realizzare le due modalità in **modo efficiente**, come stiamo per vedere

# Esercizio

- Considerando che:
  - La **compilazione condizionale** può riguardare parti qualsiasi di un file sorgente
  - Le direttive **#define** stesse sono parti del codice
- Utilizzare la compilazione condizionale per permettere in modo efficiente di **abilitare/disabilitare** tutti i messaggi di tracing in *somma_max_tracing_liv.cc*

  ed in generale di passare dalla modalità di debug alla compilazione normale

# Soluzione

- *somma_max_tracing_liv_cond.cc*

# Passaggio macro al gcc

- Consideriamo l'opzione

  **-D nome[=definizione]**

  nella riga di comando con cui si invoca il **gcc/g++** per compilare dei file
- In tutti i file passati come argomento al **gcc/g++** da quella riga di comando, l'effetto sarà equivalente all'aver aggiunto nel sorgente la direttiva

  **#define nome [definizione]**

  - Provare con *esempio_comp_condiz.cc e somma_max_tracing_liv_cond.cc*

# Nota conclusiva

- Come si vede questo è un modo **estremamente pratico** per compilare il programma in una modalità o nell'altra
- Ovviamente è importante avere chiaro che **modalità di debug non significa necessariamente solo tracing**, ma tutte le aggiunte ed i controlli che si possono ritenere necessari
- Vedremo ad esempio presto l'**uso degli invarianti e delle asserzioni nel defensive programming**

# Esercizi

Mediante tracing trovare eventuali errori nei seguenti programmi:

- *cerca_elem.cc*
- *estrai_stampa_valori.cc*