



Testing & Debugging

Laboratorio di Informatica





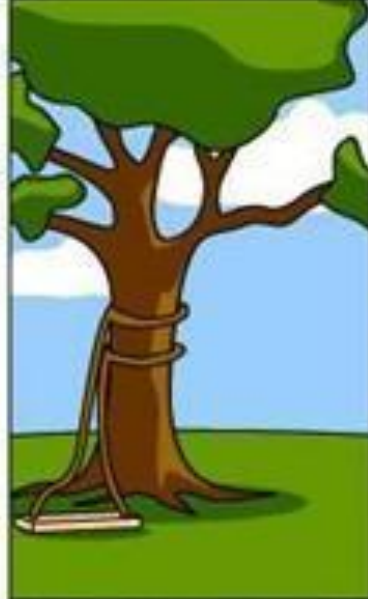
How the customer explained it



How the Project Leader understood it



How the Analyst designed it



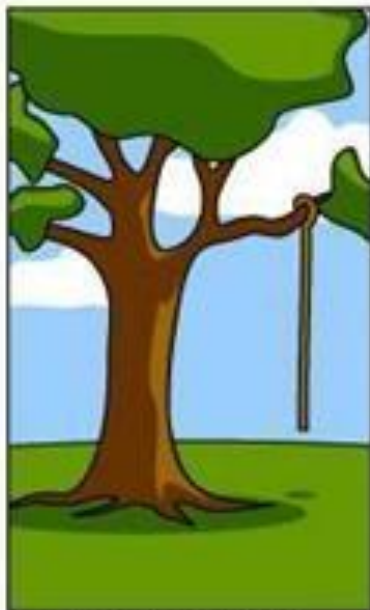
How the Programmer wrote it



How the Business Consultant described it



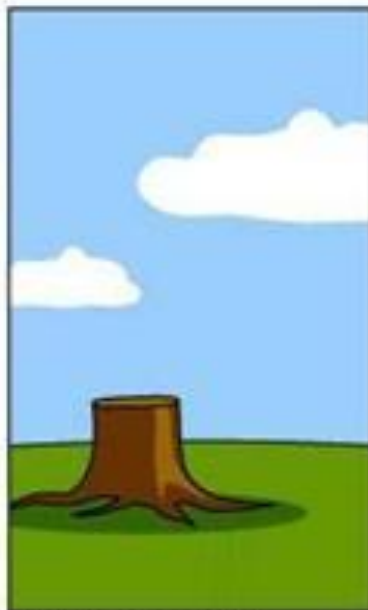
How the project was documented



What operations installed



How the customer was billed



How it was supported




What the customer really needed

i bug

- **Bug** = errore sintattico o semantico presente in un programma
- **Debug** = processo di riconoscimento e rimozione dei bug
- **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*

origine della parola "bug"

9/9

0800 Antan started
1000 " stopped - antan ✓
1300 (032) MP-MC 1.4826000 9.037 847 025
(033) PRO 2 2.13047645 9.037 846 995 correct
convert 2.13067645 4.615925059(-2)
Relays 6-2 in 033 failed special speed test
in relay "11.00 test."
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545  Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1600 Antan started.
1700 closed down.

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». Questo registro è conservato presso lo Smithsonian National Museum of American History.

fonte: Wikipedia

testing vs. debugging

- Il testing è una fase di verifica sistematica della correttezza di un software.
 - ◆ *Stima della correttezza*
 - ◆ *Parte integrante dei processi di sviluppo del sw*
- Il debugging è un processo atto a scovare la causa di un errore.

E. Dijkstra: Il testing può solo dimostrare la presenza di bug, ma non la loro assenza

- Il testing è parte prevalente nelle metodologie agili

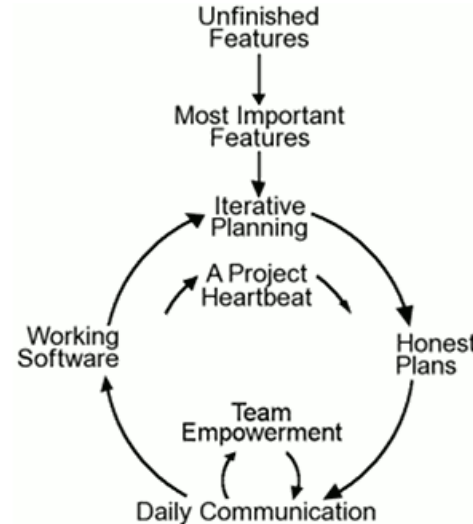
The first Extreme Programming project was started March 6, 1996. Extreme Programming is one of several popular [Agile Processes](#). It has already been proven to be very successful at many companies of all different sizes and industries world wide.

Extreme Programming is successful because it stresses customer satisfaction. Instead of delivering everything you could possibly want on some date far in the future this process delivers the software you need as you need it. Extreme Programming empowers your developers to confidently respond to changing customer requirements, even late in the life cycle.

Extreme Programming emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. Extreme Programming implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

Extreme Programming improves a software project in five essential ways; communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by [testing](#) their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. Every small success deepens their respect for the unique contributions of each and every team member. With this foundation Extreme Programmers are able to courageously respond to changing requirements and technology.


The most surprising aspect of Extreme Programming is its [simple rules](#). Extreme Programming is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces



make no sense, but when combined together a complete picture can be seen. The rules may seem awkward and perhaps even naive at first, but are based on sound [values](#) and principles.

Our rules set expectations between team members but are not the end goal themselves. You will come to realize these rules define an environment that promotes team collaboration and empowerment, that is your goal. Once achieved productive teamwork will continue even as rules are changed to fit your company's specific needs.

This [flow chart](#) shows how Extreme Programming's rules work together. Customers enjoy being partners in the software process, developers actively contribute regardless of experience level, and managers concentrate on communication and relationships. Unproductive activities have been trimmed to reduce costs and frustration of everyone involved.

Take a [guided tour](#) of Extreme Programming by following the trail of little  buttons, starting here.

“...They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested...”

Verifica delle condizioni limite

- La maggior parte dei bug si verificano in corrispondenza dei limiti
 - ◆ *Cicli: cosa succede se il numero di cicli è 0?*
 - ◆ *Array: cosa succede se si tenta di colmare un array?*
 - ◆ *Input: cosa succede se l'input acquisito è nullo?*
 - ◆ *Stream: cosa succede se si accede a un file inesistente, un disco pieno, una connessione interrotta, etc.?*
- Approccio: ogni volta che si scrive un blocco di codice significativo (ciclo, condizione, input), testarne le condizioni limite.
 - ◆ *Occorre immaginare tutte le possibili condizioni limite e documentarle*

Esempio

Questo programma tenta di leggere una sequenza di caratteri da un file e li memorizza in un array fino a quando viene letta una newline o si raggiunge la dimensione massima $MAX > 0$

```
int i = 0;  
char s[MAX];  
while ((s[i] = fgetc(file)) != '\n' && i < MAX-1)  
    i++;  
s[--i] = '\0';
```


Esempio

Quali sono le condizioni limite?

1. L'input è vuoto (“\n”)
2. $MAX == 1$
3. L'input ha una lunghezza pari a MAX
4. L'input ha una lunghezza maggiore di MAX
5. L'input non contiene una newline (se possibile)

Esempio

Riscrivendo il codice usando uno stile più leggibile, si eviterebbe l'errore del codice precedente

```
int i = 0;
char s[MAX];
int stop = 0;
while (!stop && i < MAX) {
    s[i] = fgetc(file);
    stop = (s[i] == '\n');
    i++;
}
s[--i] = '\0';
```

Esercizio

```
/**  
 * Restituisce la media aritmetica di un array.  
 *  
 * @param a double di len_a elementi; len_a>0  
 * @return media aritmetica degli elementi dell'array  
 */  
double avg(double a[], int len_a) {  
    int i;  
    double sum = 0.0;  
    for (i=0; i < len_a; i++) {  
        sum += a[i];  
    }  
    return sum / len_a;  
}
```

Programmazione difensiva

Aggiunta di codice per casi “impossibili”

```
if (age < 0 || age > MAX_AGE) {  
    range = "???";  
} else if (age <= 18) {  
    range = "Teenager"  
} ...
```

Controllo dei valori di errore restituiti

Se una funzione restituisce dei valori di errore questi vanno verificati dal chiamante

```
range = num_to_range(age);  
if (strcmp(range,"???")==0) {  
    /* errore */  
    ...  
} else {  
    ...
```

Testing sistematico

- Verifica incrementale
- Verifica bottom-up
- Verifica dei risultati attesi
- Verifica della copertura

Verifica incrementale

- Test di pari passo con l'implementazione
- Test di unità elementari
 - ◆ *Una procedura o funzione*
 - ◆ *Un blocco significativo di procedura*

Verifica bottom-up

→ Testare prima:

- ◆ *Le parti (componenti/casi) più semplici*
- ◆ *Le parti più frequentemente utilizzate*

→ Esempio: Ricerca binaria

- ◆ *Ricerca in un array vuoto*
- ◆ *Ricerca in un array con un solo elemento*
- ◆ *Ricerca di un elemento minore di quello presente*
- ◆ *Ricerca di un elemento uguale a quello presente*
- ◆ *Ricerca di un elemento maggiore di quello presente*
- ◆ *Ricerca in un array con due elementi*
 - 5 combinazioni possibili
- ◆ *Ricerca in presenza di elementi ripetuti*
- ◆ *Ricerca in presenza di elementi contigui e non contigui*

Verifica dei risultati attesi

- Per ogni test, occorre conoscere il risultato atteso
- Questo è ovvio per molti casi, ma non per tutti
 - ◆ *Testare un compilatore*
 - Compilare uno o più programmi formalmente corretti e testarli
 - ◆ *Testare un programma di calcolo numerico*
 - Verificare i limiti dell'algoritmo
 - Verificare proprietà note
 - Testare problemi con risultati già noti
 - Analisi statistiche
 - ◆ *Testare un programma grafico/multimediale*
 - Uso di strumenti di image editing
 - Analisi statistiche

Verificare la copertura dei test

→ I test devono garantire che ogni istruzione sia eseguita almeno una volta

- ◆ *Rami then-else*

- ◆ *Tutti i case di una switch*

- ◆ *Esecuzione dei cicli*

- 0 volte, 1 volta, numero max. di volte, numero max. di volte – 1

→ Classi di equivalenza

- ◆ *Es. $n \in]min, max[, n < min, n > max, n = min, n = max$*

→ L'analisi del codice può aiutare a individuare gli input che consentono di coprire tutto il codice.



CUnit

introduzione



Unit test

→ Cos'è?

- ◆ *Tecnica di progetto e sviluppo del software*

→ A cosa serve?

- ◆ *A ottenere evidenza che le singole unità software sviluppate siano corrette e pronte all'uso*

→ Unità software?

- ◆ *In un linguaggio procedurale come il C una unità può essere un programma, una funzione, ecc.*

→ Come si fa?

- ◆ *Si scrivono degli unit test (o casi di test) rappresentanti una sorta di “contratto scritto” che la porzione di codice testata deve soddisfare*

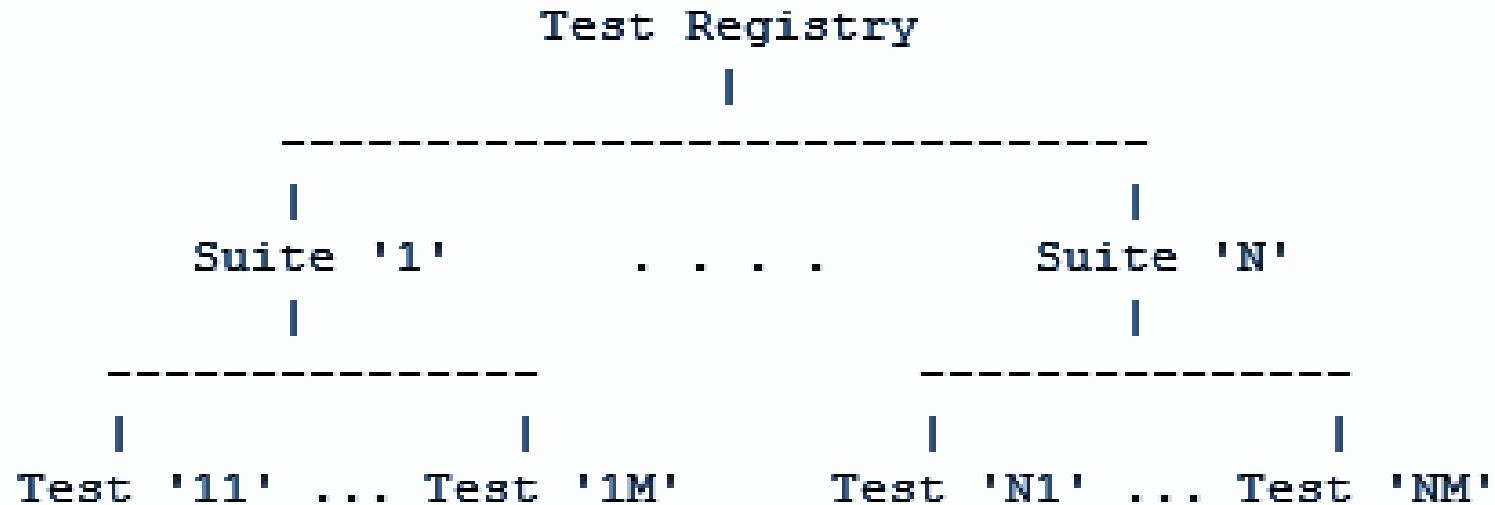
xUnit test framework

- In principio fu JUnit per Java
 - ◆ *Creato da Kent Beck e Erich Gamma*
 - ◆ *“Never in the field of software development have so many owed so much to so few lines of code” (M. Fowler a proposito di JUnit)*
- E' stato portato verso innumerevoli altri linguaggi (C/C++, C#, PHP, Python, JavaScript, Ruby, ...) dando vita all'ecosistema dei framework di tipo xUnit
- Ha dato vita al Test-Driven Development (TDD – sviluppo guidato dal test)

CUnit

- Framework di unit test per il linguaggio C
- Home page: <http://cunit.sourceforge.net/index.html>
- Libreria che va inclusa in ogni progetto Eclipse CDT che intende avvalersene
- Guida di installazione disponibile qui:
<http://collab.di.uniba.it/fabio/guide/>

Struttura del framework CUnit



- Il **framework** esegue automaticamente tutte le **test suite** inserite nel **test registry**
- Ogni **test suite** è composta da uno o più **test method** logicamente correlati

Es. suite per testare tutti i metodi di un particolare modulo

Method under test

→ Programma da testare

- ◆ *Costituito da diversi file .c (detti moduli) contenenti diverse funzioni e/o procedure*

- `funz_1(), ..., funz_n(), proc_1(), ..., proc_m()`
- Queste funzioni e procedure sono detti methods under test

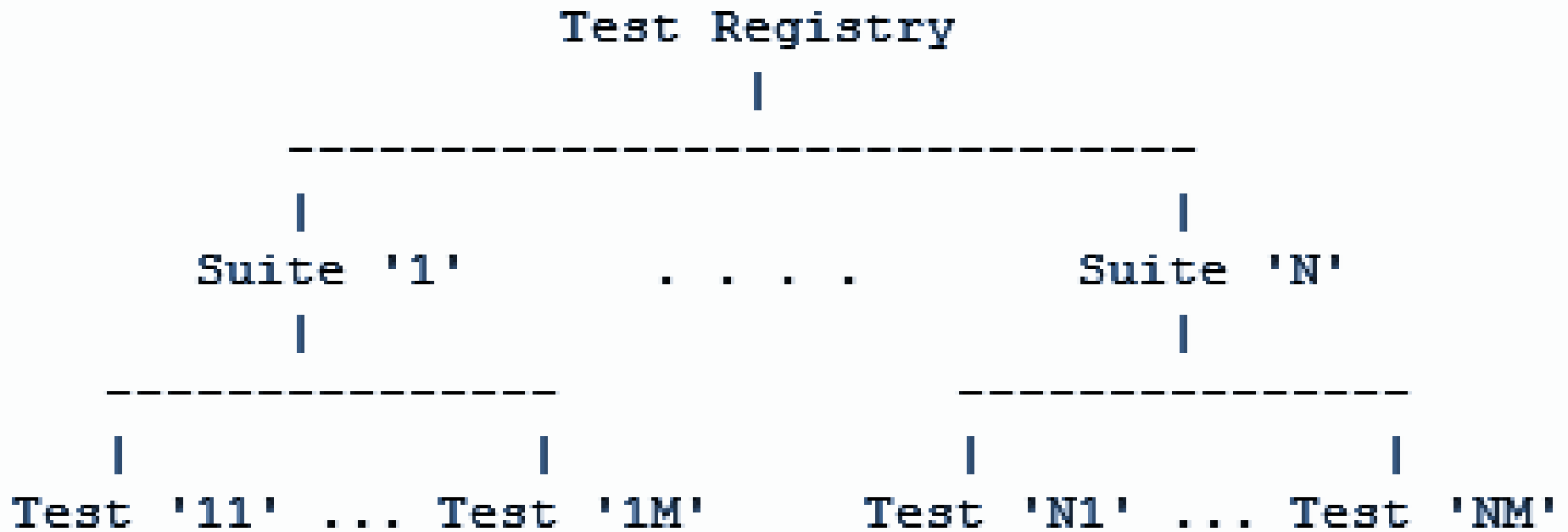
→ Per ciascun metodo da testare occorre scrivere almeno un **test method**:

- ◆ *Ciascun metodo di test va chiamato `test_xyz()`*

- `test_funz_1(), ..., test_funz_n(), test_proc_1(), ..., test_proc_m()`

→ Un metodo di test verifica la presenza di errori nel corrispettivo metodo sotto test

- ◆ *Errore \equiv comportamento diverso da quello atteso*



Attenzione: **L'ordine di inserimento ha importanza!**

- Le test suite sono eseguite nello stesso ordine di inserimento nel registry
- I test method sono eseguiti nello stesso ordine di inserimento nella suite

Ciclo di unit test

Sequenza tipica di uso di un framework di unit test, incluso CUnit:

1. Scrivi tutti i test method necessari
2. Crea il test registry
3. Crea la test suite e aggiungila al test registry
4. Aggiungi i test method alle test suite definite
5. Se necessario, ripeti i passi 3-4 per un'altra suite
6. Esegui il test registry
7. Pulisci il test registry

Procedure e Funzioni in C

un breve ripasso

Procedure e funzioni

- Sono istruzioni non primitive per risolvere parti specifiche di un problema: i **sottoprogrammi** (o **metodi**)
- Sono realizzate mediante la definizione di unità di programma (sottoprogrammi) distinte dal programma principale (main)
- Rappresentano nuove istruzioni/operatori che agiscono sui dati utilizzati dal programma
- Sono definite a partire da una sequenza di istruzioni primitive e altre procedure/funzioni

Sottoprogrammi: funzioni e procedure

→ Tutti i linguaggi di alto livello offrono la possibilità di utilizzare funzioni e procedure mediante:

- ◆ *costrutti per la **definizione** di sottoprogrammi*
- ◆ *meccanismi per la **chiamata** a sottoprogrammi*

→ Nei linguaggi ad alto livello funzioni e procedure sono molto utili per raggiungere:

- ◆ *Astrazione*
- ◆ *Riusabilità*
- ◆ *Modularità (strutturazione)*
- ◆ *Leggibilità*

Funzioni e procedure: definizione

Nella fase di definizione di una funzione o procedura si stabilisce:

→ Un **identificatore** del sottoprogramma

◆ *(cioè il nome da usare per chiamare/invocare lo stesso)*

→ Un **corpo** del sottoprogramma

◆ *(cioè, l'insieme di istruzioni che sarà eseguito ogni volta che il sottoprogramma sarà chiamato)*

→ Una lista di **parametri formali**

◆ *(cioè come avviene la comunicazione tra l'unità di programma che usa il sottoprogramma ed il sottoprogramma stesso)*

Esempio di funzione e procedura in C

Funzione

```
// prototipo
int sum(int a, int b);

// dichiarazione
int sum(int a, int b) {
    int c = a + b;
    return c;
}

// chiamata
int main(void) {
    int op1, op2, result;
    ...
    result = sum(op1, op2);
    return 0;
}
```

Procedura

```
// prototipo
void print(int a);

// dichiarazione
void print(int a) {
    printf("%d", a);
}

// chiamata
int main(void) {
    int value;
    ...
    print(value);
}
```

Funzioni e procedure: differenze

→ Funzione

- ◆ *Operatore non primitivo*
- ◆ *Permette di definire nuovi operatori complessi da affiancare a quelli primitivi*
- ◆ *Restituisce un valore di ritorno (mediante return)*

→ Procedura

- ◆ *Istruzione non primitiva*
- ◆ *E' attivabile in un qualunque punto del programma in cui può comparire un'istruzione*
- ◆ *Non restituisce un valore di ritorno (mediante return)*

Funzioni e procedure: parametri

- I parametri costituiscono il mezzo di comunicazione tra unità chiamante e unità chiamata
 - ◆ *Supportano lo scambio di informazioni tra chiamante e sottoprogramma*
- Si differenziano in
 - ◆ *Parametri formali (specificati nella definizione)*
 - ◆ *Parametri attuali (specificati nella chiamata)*
- Parametri attuali e formali devono corrispondersi in numero, posizione e tipo

Parametri formali e parametri attuali

→ Parametri formali

- ◆ *Sono quelli specificati nella definizione del sottoprogramma*
- ◆ *Sono in numero prefissato e a ognuno di essi viene associato un tipo*
- ◆ *Le istruzioni del corpo del sottoprogramma utilizzano i parametri formali*

→ Parametri attuali

- ◆ *Sono i valori effettivamente forniti dall'unità chiamante al sottoprogramma all'atto della invocazione*

Associazione tra parametri attuali e parametri formali

- Il passaggio di parametri può avvenire in due modi:
 - ◆ *Per valore*
 - ◆ *Per indirizzo (o riferimento) ← lo vedremo più avanti*
- Il C di default adotta il passaggio per valore
 - ◆ *Il valore dei parametri è copiato nello stack*
 - ◆ *Il passaggio per riferimento si ottiene memorizzando nello stack l'indirizzo (puntatore) in cui è allocata una variabile*
 - ◆ *Il passaggio per valore è anche più “sicuro”*



.... Di ritorno a CUnit



Aggiungere le librerie CUnit a un progetto

Guida di installazione e configurazione disponibile alla sezione dispense sul sito del corso

<http://collab.di.uniba.it/fabio/guide/>

Includere le librerie di CUnit nei file di test di un progetto

Quando preparate il modulo con i metodi di test ricordate di includere i file header in questo modo:

...

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include "CUnit/Basic.h"
```

...

Ciclo di Unit Test

1.Scrivi tutti i test method necessari

2.Crea il test registry

3.Crea la test suite e aggiungila al test registry

4.Aggiungi i test method alle test suite definite

5.Se necessario, ripeti i passi 3-4 per un'altra suite

6.Esegui il test registry

7.Pulisci il test registry

Scrivere un test method

- Un metodo di test in CUnit si presenta sempre nella forma di **procedura senza parametri**
 - ◆ `void test_xyz(void)`
- Un metodo di test è un contratto che stabilisce i vincoli che devono essere soddisfatti dal software
 - ◆ *I vincoli sono stabiliti attraverso delle asserzioni*
 - ◆ *Un'asserzione in un linguaggio di programmazione è una funzione che verifica una condizione logica e restituisce:*
 - Vero, se l'asserzione è rispettata
 - Falso, altrimenti

Assertzioni di base (CUnit)

Assertzione	Significato
CU_ASSERT (<i>int espressione</i>) CU_TEST (<i>int espressione</i>)	Assertisce che espressione è TRUE (diverso da 0)
CU_ASSERT_TRUE (<i>valore</i>)	Assertisce che valore è TRUE (diverso da 0)
CU_ASSERT_FALSE (<i>valore</i>)	Assertisce che valore è FALSE (uguale a 0)
CU_ASSERT_EQUAL (<i>reale, atteso</i>)	Assertisce che reale == atteso
CU_ASSERT_NOT_EQUAL (<i>reale, atteso</i>)	Assertisce che reale != atteso
CU_ASSERT_STRING_EQUAL (<i>reale, atteso</i>)	Assertisce che le stringhe reale e atteso coincidono
CU_ASSERT_STRING_NOT_EQUAL (<i>reale, atteso</i>)	Assertisce che le stringhe reale e atteso differiscono

Esempio di test method per la funzione max(a,b)

```
void test_max(void) {  
  
    CU_ASSERT_EQUAL(max(0,2), 2);  
  
    CU_ASSERT_TRUE(max(0,-2) == 0);  
  
    CU_TEST(max(2,2) == 2);  
  
    // questa asserzione è sbagliata e fallisce  
    CU_ASSERT_TRUE(max(5,6) == 2);  
}
```

Esempio di metodo di test per la funzione factorial(x)

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

```
void test_factorial(void) {  
    // fallisce  
    CU_ASSERT_EQUAL(factorial(4), 12);  
  
    CU_ASSERT(factorial(3) == 6);  
  
    CU_TEST(factorial(6) == 720);  
  
}
```

Ciclo di Unit Test

~~1. Scrivi tutti i test method necessari~~

2. Crea il test registry

3. Crea la test suite e aggiungila al test registry

4. Aggiungi i test method alle test suite definite

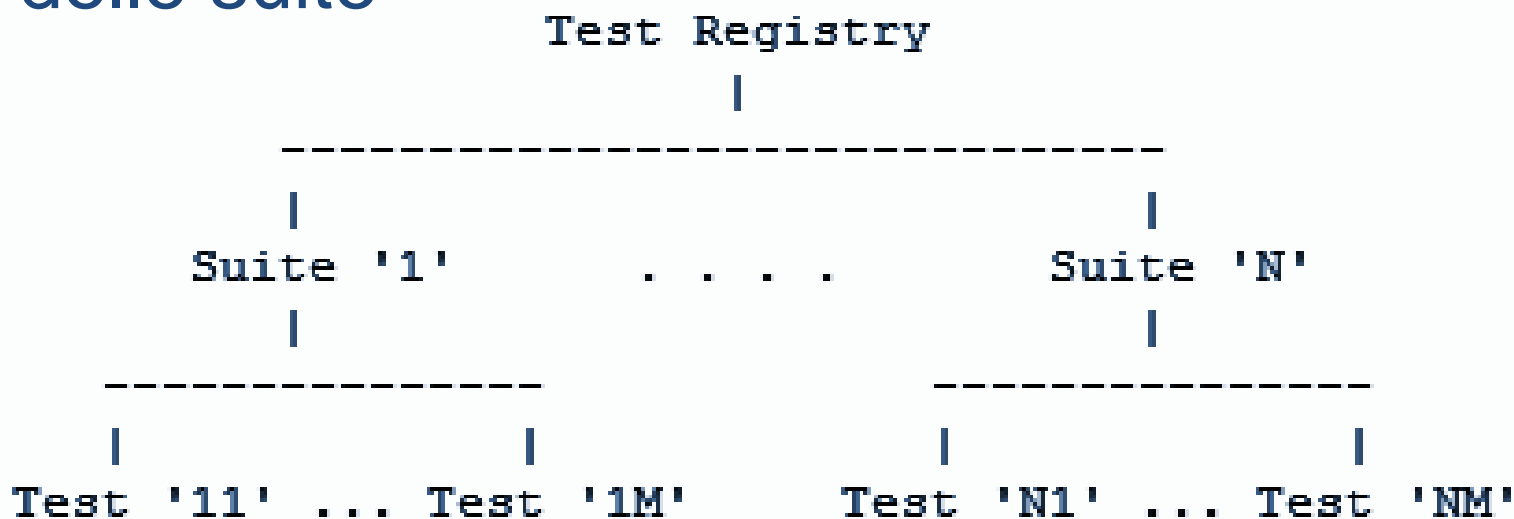
5. Se necessario, ripeti i passi 3-4 per un'altra suite

6. Esegui il test registry

7. Pulisci il test registry

Il Test Registry

- Raccoglie tutte le test suite
- Quando si esegue un Test Registry si eseguono tutte le suite al suo interno e, di conseguenza, tutti i test method all'interno delle suite



Inizializzazione del Test Registry

```
00
61 // *****
62 //  TEST di UNITA'
63
64 int main() {
65     /* inizializza registro - e' la prima istruzione */
66     CU_initialize_registry();
67
```

L'inizializzazione del test registry è la prima operazione da effettuare

Ciclo di Unit Test

1. ~~Scrivi tutti i test method necessari~~

2. ~~Crea il test registry~~

3. Crea la test suite e aggiungila al test registry

4. ~~Aggiungi i test method alle test suite definite~~

5. ~~Se necessario, ripeti i passi 3-4 per un'altra suite~~

6. ~~Esegui il test registry~~

7. ~~Pulisci il test registry~~

Test Suite

- Una test suite è definita da:
 - ◆ *Una descrizione testuale*
 - ◆ *Una procedura di inizializzazione (init)*
 - ◆ *Una procedura di pulitura (clean)*
- Le test suite definite vengono aggiunte al test registry (l'ordine è rilevante!)

```
/* Aggiungi le suite al test registry */  
CU_pSuite pSuite_A = CU_add_suite("Suite_A", init_suite_default, clean_suite_default);  
CU_pSuite pSuite_B = CU_add_suite("Suite_B", init_suite_default, clean_suite_default);
```

- Di default inizializzazione e pulitura sono procedure vuote.

Inizializzazione e pulizia delle suite

- Le test suite devono essere inizializzate e ripulite prima e dopo l'uso
 - ◆ *I metodi non sono forniti da CUnit ma devono essere scritti dal programmatore*
- Perché?
 - ◆ *Perché devono liberare le risorse allocate **specificatamente** per eseguire il caso di test*
 - Es. file, connessioni, etc.

Inizializzazione e pulizia

```
// Alloca tutte le risorse necessarie all'esecuzione  
// dei test
```

```
int init_suite_default(void) {  
    return 0; // tutto ok  
}
```

```
// dealloca tutte le risorse allocate all'inizializzazione
```

```
int clean_suite_default(void) {  
    return 0; // tutto ok  
}
```

Ciclo di Unit Test

- 1. ~~Scrivi tutti i test method necessari~~*
- 2. ~~Crea il test registry~~*
- 3. ~~Crea la test suite e aggiungila al test registry~~*
- 4. Aggiungi i test method alle test suite definite**
- 5. Se necessario, ripeti i passi 3-4 per un'altra suite**
- 6. Esegui il test registry*
- 7. Pulisci il test registry*

Test method e test suite

→ Un test method viene aggiunto ad una test suite specificando:

- ◆ il puntatore alla suite
- ◆ una descrizione testuale del test
- ◆ il puntatore al test method

```
/* Aggiungi i test alle suite
 * NOTA - L'ORDINE DI INSERIMENTO E' IMPORTANTE
 */
CU_add_test(pSuite_A, "test of f1()", test_f1);
CU_add_test(pSuite_A, "test of f3()", test_f3);

CU_add_test(pSuite_B, "test of f4()", test_f4);
CU_add_test(pSuite_B, "test of f2()", test_f2);
```

L'ordine dei test nelle suite è rilevante!

Ciclo di Unit Test

- ~~1. Scrivi tutti i test method necessari~~
- ~~2. Crea il test registry~~
- ~~3. Crea la test suite e aggiungila al test registry~~
- ~~4. Aggiungi i test method alle test suite definite~~
- ~~5. Se necessario, ripeti i passi 3-4 per un'altra suite~~
- 6. Esegui il test registry**
- 7. Pulisci il test registry*

Registrazione ed esecuzione dei test

- La procedura CU_basic_run_tests esegue tutte le suite del registry e mostra i risultati

```
/* Esegue tutti i casi di test con output sulla console */  
CU_basic_set_mode(CU_BRM_VERBOSE);  
CU_basic_run_tests();
```

È possibile impostare il livello di “verbosità” dell’output

Ciclo di Unit Test

- ~~1. Scrivi tutti i test method necessari~~
- ~~2. Crea il test registry~~
- ~~3. Crea la test suite e aggiungila al test registry~~
- ~~4. Aggiungi i test method alle test suite definite~~
- ~~5. Se necessario, ripeti i passi 3-4 per un'altra suite~~
- ~~6. Esegui il test registry~~
- 7. Pulisci il test registry**

Pulire il registry

→ Pulizia – dopo aver eseguito tutti i test nel registry

◆ *Procedura void CU_cleanup_registry(void)*

```
/* Pulisce il registry e termina lo unit test */  
CU_cleanup_registry();  
  
return CU_get_error();
```

→ Il **main()** termina con la return dell'eventuale codice di errore di CUnit

Esercitazione 1

CUnit

Link all'esercitazione

<http://goo.gl/VYfhsN>

Implementazione di una serie di funzioni e testing di queste con l'utilizzo di CUnit

template di CUnit disponibile a

<http://goo.gl/uevMHu>

Debugging



Ariane 5 Flight 501

testing vs. debugging

- Il testing è una fase di verifica sistematica della correttezza di un software.
- **Il debugging è un processo atto a scovare la causa di un errore.**
 - ◆ *è un processo costoso, dai tempi non prevedibili*
 - ◆ *l'esperienza è importante*
 - ◆ *gli strumenti possono velocizzare il debugging*

Supporto del compilatore

→ Molti compilatori emettono dei “warning”, cioè dei messaggi di avvertimento

- ◆ *if (a=0) ...*
- ◆ *x = x*
- ◆ *nessun return*
- ◆ *codice orfano*
- ◆ *condizioni tautologiche*
- ◆ *...*

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*
- Scrivere codice leggibile aiuta al backward reasoning e, quindi, a localizzare i bug

Pattern familiari

- Riconoscere variazioni rispetto a “modelli” (pattern) di codice familiari

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

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

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

Sviluppo incrementale

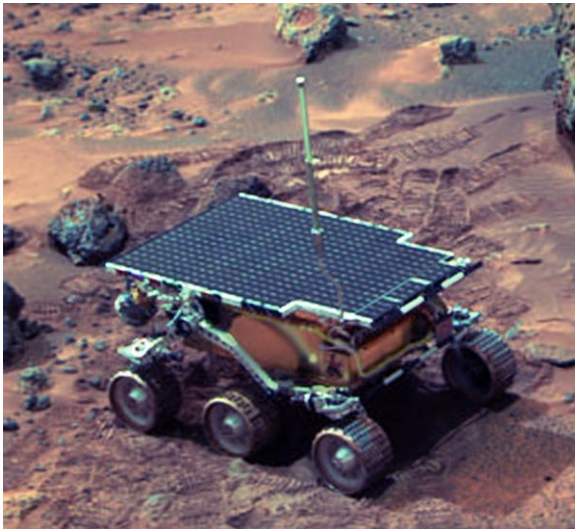
- Testare le procedure man mano che vengono sviluppate
 - ◆ *Se i test all'istante t hanno successo ma falliscono all'istante $t+1$, allora molto probabilmente i bug si annidano nel codice sviluppato tra t e $t+1$*
- La progettazione modulare del codice aiuta a individuare meglio la posizione dei bug

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”*
- Una buona progettazione del codice riduce la ridondanza e, quindi, la possibilità di bug duplicati

Non rimandare il debugging

- 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.*



“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)

Leggere e spiegare il codice

- Leggere il codice e comprenderne il significato
 - ◆ *Il codice è un pezzo di conoscenza che deve essere compreso dalla macchina e da chi la programma*
 - ◆ *La leggibilità del codice è fondamentale*
- Spiegare ad altri il codice aiuta a ridurre “bias” cognitivi

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*
- ◆ *...*

Divide et impera

- Individuare le condizioni minimali che rendono manifesto un bug
 - ◆ *es. il più piccolo array, la stringa più breve*
 - Test dei casi limite è fondamentale
 - ◆ *Le condizioni minimali possono facilitare la localizzazione di un bug*
 - ◆ *Se il bug non si manifesta in un caso limite, provare mediante dimezzamenti successivi dell'input*
 - Ricerca binaria sulla lunghezza dell'input

Ricerca di regolarità

- Alcuni bug si presentano con regolarità, ma non sempre
- In questo caso, occorre capire il modello (“pattern”) che genera la regolarità
 - ◆ *Es. Un editor di testi salta la visualizzazione di alcuni caratteri*
 - ◆ *L’analisi del testo mostra che i caratteri saltati sono sempre intervallati da 1023 caratteri stampati*
 - Regolarità: 1 carattere saltato ogni 1023
 - ◆ *L’analisi del codice rivela che gli array che memorizzano le stringhe sono da 1024 byte*
 - 1023 caratteri + ‘\0’ → BUG

Stampe ausiliarie

- Per seguire l'esecuzione di un programma può essere utile introdurre stampe ausiliarie
 - ◆ *Valido soprattutto per situazioni che non possono essere tracciate da un debugger*
 - es. sistemi distribuiti, programmi paralleli, etc.
- Le stampe ausiliarie devono necessariamente essere eliminate dopo aver scovato il bug
 - ◆ *Rischio di violazione delle specifiche*
 - ◆ *Possono essere commentate anziché eliminate*
- Per situazioni complesse, si possono usare strumenti di logging

Altre tecniche

- Visualizzazioni grafiche
- Test statistici
- Strumenti di analisi di testo
 - ◆ *grep*
 - ◆ *diff*
 - ◆ ...

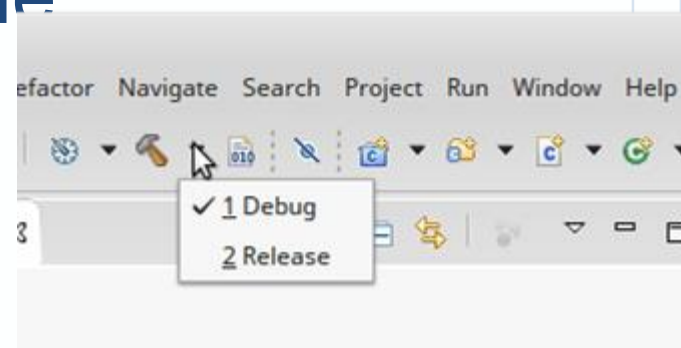
Debugger

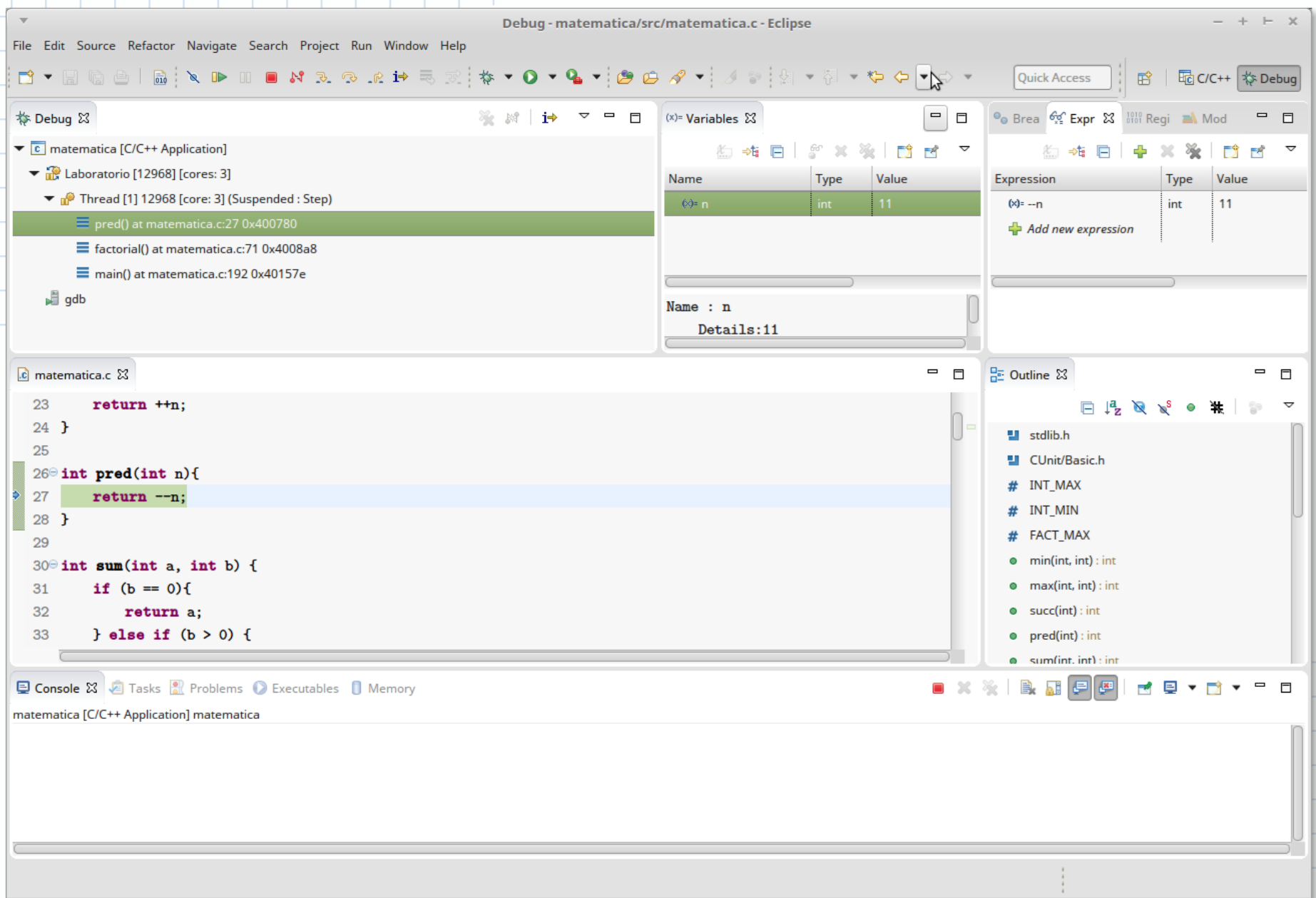
- Un debugger guarda “dentro” il programma durante l'esecuzione
 - ◆ *Tracing del programma*
 - ◆ *Visualizzazione del contenuto delle variabili*
 - ◆ *Valutazione dinamica di espressioni*
 - ◆ *Breakpoint, anche condizionali*
 - ◆ *Stack trace*
 - ◆ *...*
- Sono strumenti molto sofisticati, abituarsi al loro uso può migliorare significativamente la produttività nella programmazione.



Compilazione per il debug

- 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



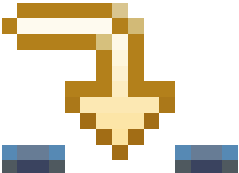
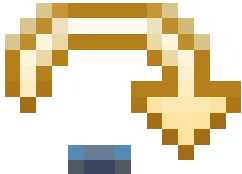



la debug perspective in Eclipse

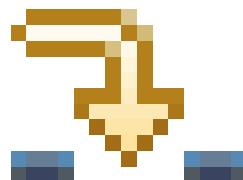
Esecuzione passo-passo

- Il debugger consente di eseguire il programma una istruzione alla volta
 - ◆ *Al termine dell'esecuzione di una istruzione, il controllo passa al debugger, che può visualizzare lo stato della macchina (variabili, stack, etc.)*
- Per velocizzare il processo di debugging, si può optare per eseguire il programma fino a un'istruzione specifica, segnalata da un **breakpoint**.

Esecuzione passo-passo

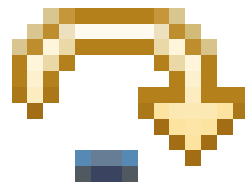
	Step into	Esegue l'istruzione corrente, e procede all'istruzione successiva che sarà effettivamente eseguita
	Step over	Esegue l'istruzione corrente, trattando le routine come istruzioni primitive.
	Step return	continua l'esecuzione fino al termine della procedura.

```
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) {
```

```
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         }
```

```

54 int product(int a, int b) {
55     if (b == 0){
56         return 0;
57     } else if (b > 0){
58         int result = 0;
59         int i;
60         for (i=0; i<b; i++){
61             result = sum(result, a);
62         }
63         return result;
64     } else {
65         return -product(a,
66     }
67 }

```



```

70 //printf("%d, ", n);
71 return (n <= 0) ? 1 : product(facto
72 }

```


Informazioni di debug: variabili

→ Le variabili visibili nell'ambito dell'istruzione corrente sono visualizzate

◆ *Nome, tipo, valore*

→ Le variabili che cambiano valore sono evidenziate

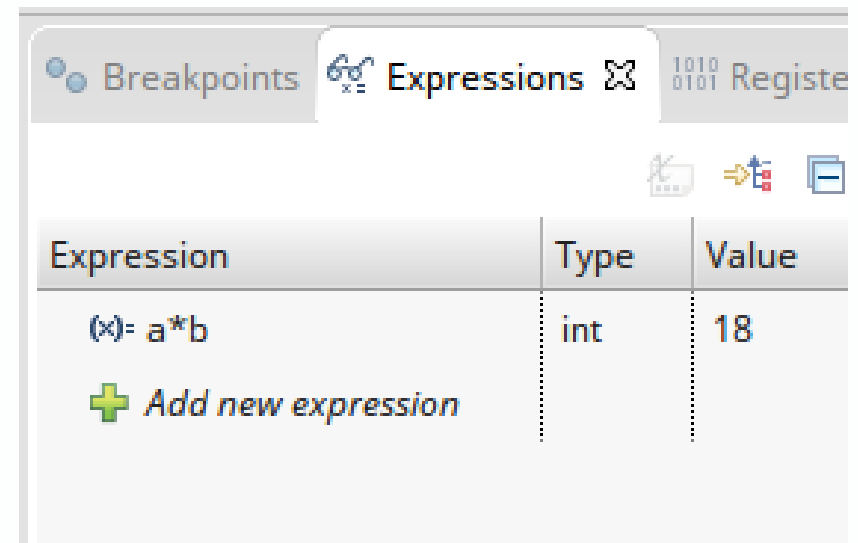
```
54 int product(int a, int b) {  
55     if (b == 0){  
56         return 0;  
57     } else if (b > 0){  
58         int result = 0;  
59         int i;  
60         for (i=0; i<b; i++){  
61             result = sum(result, a);  
62         }  
63         return result;  
64     } else {
```

(x)= Variables

Name	Type	Value
(x)= a	int	3
(x)= b	int	6
(x)= result	int	0
(x)= i	int	1

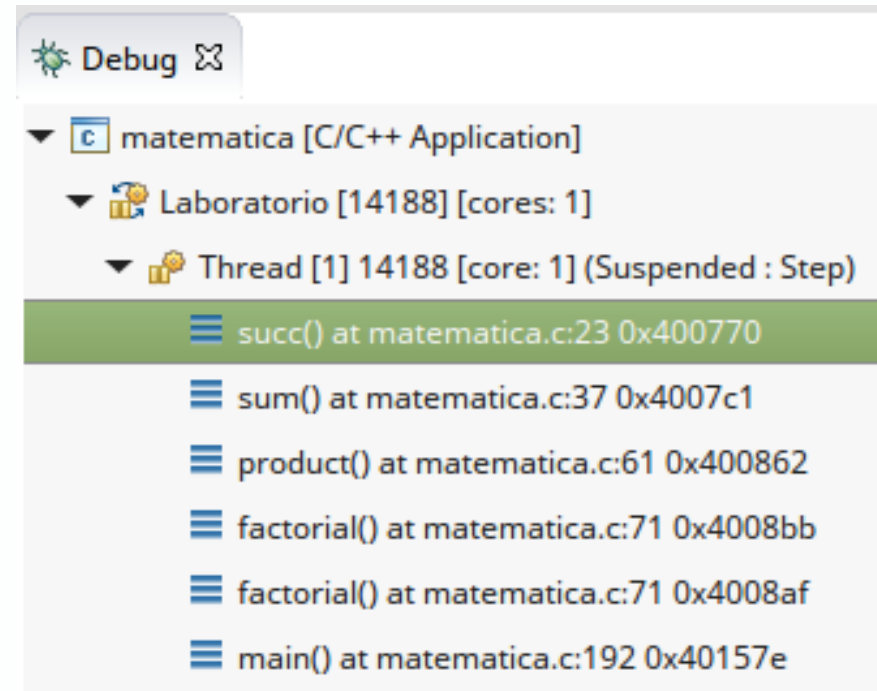
Informazioni di debug: espressioni

→ Un'espressione è un pezzo ben formato di codice (snippet) che può essere valutato per produrre un risultato



Informazioni di debug: stack trace

- Visualizza la pila delle chiamate
- Si può selezionare un elemento della pila per conoscerne lo stato corrente

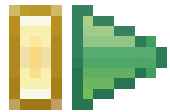


Breakpoint

- Un breakpoint interrompe il flusso di esecuzione su una linea selezionata
- ◆ I breakpoint possono essere inseriti o rimossi
 - ◆ I breakpoint inseriti possono essere attivati o disattivati

```
60     for (i=0; i<b; i++){  
61         result = sum(result, a);  
62     }  
63     return result;  
64 } else {  
65     return -product(a, -b);  
66 }
```

Resume & 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
 - Utile quando si scova un programma
- ◆ *Attenzione: i programmi non terminati rimangono in esecuzione per il sistema operativo*
 - Occupazione inutile di memoria
 - Problemi per la compilazione

Breakpoint condizionali

→ I breakpoint possono interrompere l'esecuzione solo quando una condizione diventa vera

◆ *Condizione: espressione booleana*

