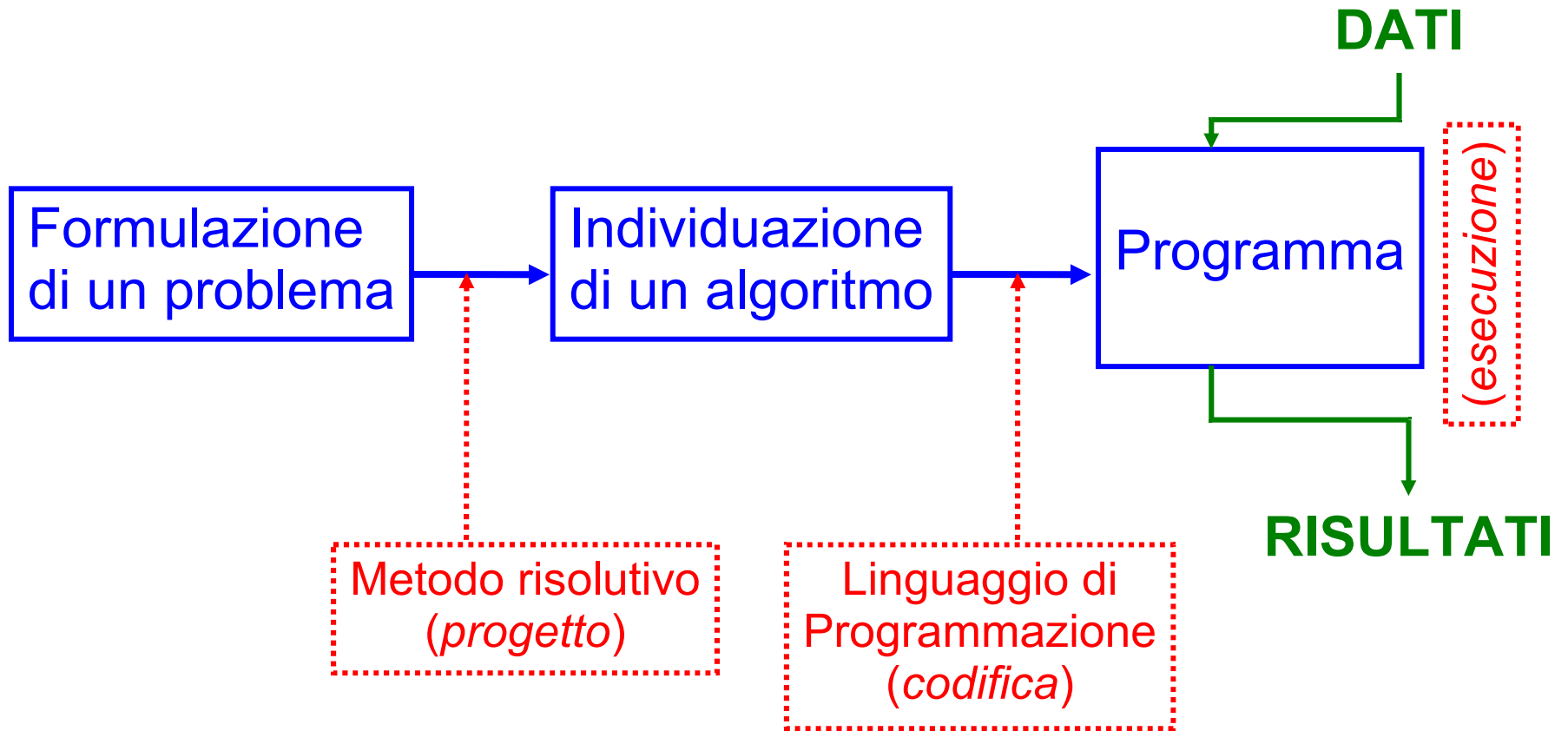


Lezione 16

Algoritmi e programmi,
traduttori e compilatori,
ambienti di sviluppo

Dal problema al programma



Proprietà di un algoritmo 1/3

- **Eseguibilità:** ogni azione deve essere *eseguibile* da parte dell'esecutore dell'algoritmo in un tempo finito
- **Non-ambiguità:** ogni azione deve essere *univocamente interpretabile* dall'esecutore
- **Terminazione:** il numero totale di azioni da eseguire, per ogni insieme di dati di ingresso, deve essere finito

Proprietà di un algoritmo 2/3

- Altre proprietà o modi di esprimere quelle già viste:
 - Un algoritmo deve
 - essere *applicabile a qualsiasi insieme di dati di ingresso* appartenenti al dominio di definizione dell'algoritmo stesso
 - essere costituito da operazioni
 - appartenenti ad un determinato insieme di operazioni fondamentali
 - non ambigue, ossia interpretabili in modo univoco qualunque sia l'esecutore (persona o "macchina") che le legge

Proprietà di un algoritmo 3/3

- ALTRE PROPRIETA'
 - Efficienza (proprietà desiderabile)
 - Numero di passi da effettuare, occupazione di memoria
 - Determinismo
 - Esistono anche algoritmi volutamente non deterministici
 - Allo scopo di raggiungere una maggiore efficienza

Algoritmi equivalenti

- Due algoritmi si dicono equivalenti quando:
 - hanno lo stesso dominio di ingresso
 - hanno lo stesso dominio di uscita
 - in corrispondenza degli stessi valori nel dominio di ingresso producono gli *stessi valori nel dominio di uscita*
- Due algoritmi equivalenti:
 - forniscono lo stesso risultato
 - ma possono avere **diversa efficienza** (numero di passi da effettuare, quantità di memoria occupata)
 - e possono essere profondamente diversi !

Costo computazionale

- L'efficienza in termini di tempo di esecuzione di un algoritmo dipende dal suo **costo computazionale**, ossia dal numero di passi da effettuare per ottenere l'obiettivo per cui è stato definito
 - Spesso chiamata un po' impropriamente anche *complessità* dell'algoritmo

Problema

- Purtroppo il numero esatto di passi elementari effettuati da un algoritmo per risolvere un dato problema può variare
 - in base all'insieme di valori di ingresso
 - Magari fino ad una certa dimensione del problema (numero N di elementi su cui si lavora) c'è un certo costo, poi il costo cambia
 - Tipicamente il tempo di esecuzione di un algoritmo può diventare un problema pratico solo quando il numero di elementi su cui l'algoritmo lavora è molto grande
 - ed in funzione di come si è definito l'insieme delle azioni elementari che si possono eseguire
 - Magari per ogni elemento su cui lavora l'algoritmo si effettua concettualmente sempre un solo passo base, ma a seconda di come si realizza il passo base, cambia il numero di operazioni

Possibile soluzione 1/2

- Per rendere la cosa esplicita abbiamo bisogno di una notazione che ci permetta di scrivere appunto che l'algoritmo fa un solo passo per ogni elemento, indipendente mente da quante operazioni costituiscono tale passo
 - In altri termini, vorremmo poter dire che l'algoritmo fa N passi in totale per risolvere il problema
- Un algoritmo che fa N passi per lavorare su N elementi, ha sicuramente un costo molto conveniente all'aumentare di N !!!
 - Quasi sempre gli N elementi vanno perlomeno letti in input, quindi N passi sono il numero minimo di passi necessari già solo per leggere i valori in ingresso!

Possibile soluzione 2/2

- Quindi, se avessimo una notazione che ci permetta di scrivere esplicitamente che l'algoritmo fa solo N passi, questo ci permetterebbe di distinguerlo bene, per esempio
 - da un algoritmo che invece deve effettuare N^2 passi in totale
 - Anche se il singolo passo necessario a questo secondo algoritmo fosse costituito da meno operazioni, al crescere di N il primo finisce per rivelarsi più efficiente

Ordine di costo 1/3

- Per misurare il costo di un algoritmo per alti valori di N , ed indipendentemente dal numero di operazioni compiute ad ogni passo si utilizza il concetto di ordine di costo
 - Vediamone la definizione prima per il solo caso del nostro algoritmo che compie solo N passi
 - Dato il numero di elementi N su cui un algoritmo lavora (dimensione del problema), si dice, per esempio, che l'algoritmo ha costo di **ordine $O(N)$** se esistono un valore N_{min} ed una costante c tali che, per tutti i valori di N maggiori di N_{min} , il numero di passi effettuati dall'algoritmo non è mai maggiore di $c * N$

Ordine di costo 2/3

- Nella notazione $O(N)$ ci sono proprio:
 - Il concetto che ci interessa cosa succede quando N è grande (cioè a partire dal momento in cui supera un certo valore minimo N_{min})
 - La costante c che ci permette di eliminare il dettaglio di quante operazioni servono per ogni passo
 - Più è alto tale numero di operazioni e più scegliamo alto il valore di tale costante
 - Scegliendo un valore alto di c , il numero totale di operazioni effettuate dall'algorithmo rimane sempre inferiore a $c * N$
 - E quindi possiamo continuare a dire che l'algorithmo è $O(N)$

Ordine di costo 2/3

- La definizione generale dell'ordine di costo di un algoritmo è la seguente:
 - Dato il numero di elementi N su cui un algoritmo lavora (dimensione del problema), l'algoritmo ha costo di ordine $O(f(N))$ se esistono un valore N_{min} ed una costante c tali che, per tutti i valori di N maggiori di N_{min} , il numero di passi effettuati dall'algoritmo non è mai maggiore di $c * f(N)$
- Al di là dei dettagli tecnici (che vedrete nel corso di **Algoritmi e Strutture Dati**), possiamo fare le seguenti importanti considerazioni su alcuni ordini di costo
 - $O(N)$: l'algoritmo effettua un numero di passi proporzionale al numero di elementi su cui lavora

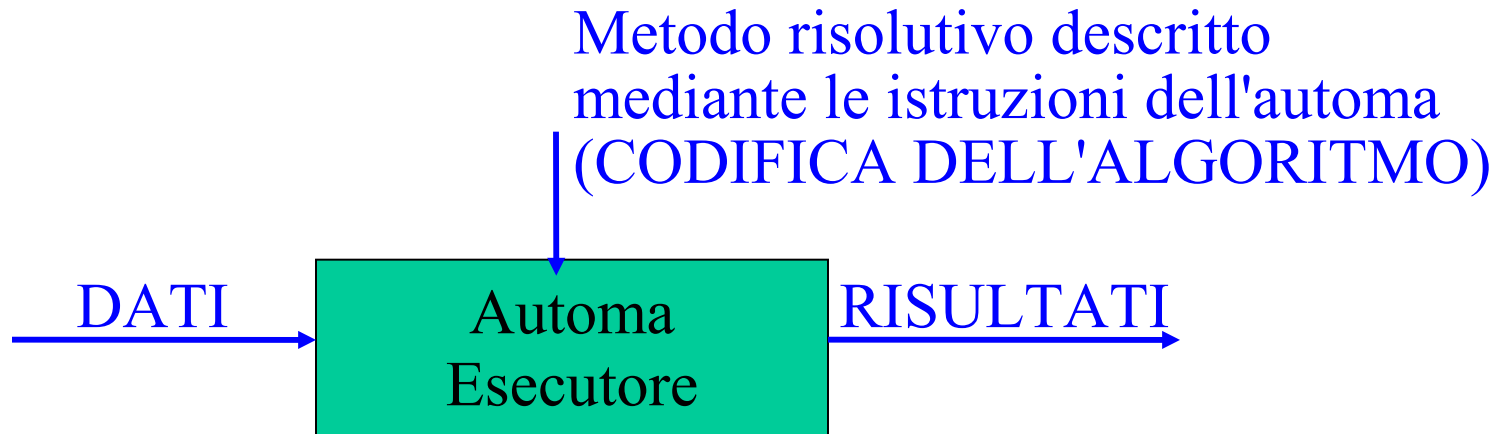
Costo computazionale 3/3

- $O(N^2)$: l'algoritmo effettua un numero di passi pari al quadrato del numero di elementi su cui lavora
- $O(N^i)$: ordine di costo polinomiale, su problemi di grosse dimensioni ogni volta che i aumenta il tempo di esecuzione diviene enormemente più lungo
- $O(a^N)$: ordine di costo esponenziale, al crescere di N il tempo di esecuzione dell'algoritmo diviene rapidamente così lungo da renderne impossibile il completamento in tempi ragionevoli
- All'estremo opposto abbiamo:
 - $O(1)$: ordine di costo costante, il numero di passi effettuati dall'algoritmo è **indipendente** dal numero di elementi su cui lavora

Esecutore ed istruzioni

- Un algoritmo è di una qualche utilità se esiste un esecutore in grado di eseguirlo
- Un esecutore, spesso chiamato **automa esecutore**, può essere istruito in modo molto efficace per eseguire un algoritmo se
 - 1) Può essere programmato mediante un insieme di istruzioni (che è in grado di eseguire)
 - 2) Tale insieme di istruzioni ha le seguenti caratteristiche:
 - Le istruzioni sono sufficienti per eseguire i passi dell'algoritmo che si vuol fare eseguire a tale esecutore
 - La sintassi e la semantica delle istruzioni sono complete e non ambigue

Automa esecutore



- Come realizzare l'automa?
 - mediante congegni meccanici
 - macchina aritmetica (1649) di Blaise Pascal
 - macchina analitica di Charles Babbage (1792-1871)
- Mediante un modello matematico (automa astratto)
 - funzionale (Hilbert, (1842-1943), Church, Kleene)
 - operativa (Turing, 1912-1954)
 - sistemi di riscrittura (Post, Markov,...)

Linguaggio di programmazione

- Un insieme di istruzioni con le precedenti caratteristiche costituisce come sappiamo un linguaggio di programmazione
- In definitiva, quando programmiamo per esempio in linguaggio C/C++, presumiamo la presenza di un esecutore in grado di eseguire appunto le istruzioni di un programma in linguaggio C/C++

Per codificare un algoritmo ...

... in un dato linguaggio di programmazione

- Bisogna conoscere la **sintassi** del linguaggio
- Bisogna conoscere la **semantica** del linguaggio
- Bisogna conoscere almeno un **ambiente di programmazione** per tale linguaggio
 - Editor, compilatore, ...

Algoritmo e programma

- Ovviamente al variare del linguaggio di programmazione, lo stesso algoritmo sarà codificato mediante un diverso programma
- Consideriamo per esempio il seguente algoritmo
 - Leggi due numeri interi e stampane la somma
- Il relativo programma sarà diverso a seconda che si usi il C++ o il C

Programma in C++

```
#include <iostream>
using namespace std ;

int main()
{
    int A, B, ris;
    cout<<"Immettere due numeri: ";
    cin>>A;
    cin>>B;
    ris=A+B;
    cout<<"Somma:"<<ris<<endl;
    return 0 ;
}
```

Programma in C

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

```
int main()
```

```
{
```

```
    int A, B, ris;
```

```
    printf("Immettere due numeri: ");
```

```
    scanf("%d", &A);
```

```
    scanf("%d", &B);
```

```
    ris=A+B;
```

```
    printf("Somma: %d\n", ris);
```

```
    return 0 ;
```

```
}
```

Macchina virtuale

- Concentrandoci ora sull'esecuzione di un programma in C/C++, in effetti noi assumiamo che esista un automa esecutore in grado di
 - eseguire le istruzioni (definizioni, dichiarazioni, istruzioni condizionali e cicliche, ...) del linguaggio C/C++
 - supportare il concetto di variabile, costante con nome, tipo di dato, funzione
- Chiamiamo **macchina virtuale** tale automa esecutore
 - Definiamo virtuale tale macchina perché, come stiamo per vedere, non esiste praticamente nessuna macchina reale (fisica) che abbia tali caratteristiche

Macchina reale: elaboratore

- Dal punto di vista reale invece, abbiamo eseguito i nostri programmi su degli elaboratori elettronici
- Ma, in accordo a quanto detto nella precedente slide, cominciamo dall'indagare sul fatto che un tipico elaboratore possa o meno eseguire direttamente un programma scritto in linguaggio C/C++
- Le azioni elementari che un elaboratore può effettuare si chiamano tipicamente **istruzioni macchina**
- Il componente di un elaboratore che si occupa dell'esecuzione di tali istruzioni è tipicamente chiamato **(micro)processore**

Linguaggio macchina 1/2

- L'insieme delle istruzioni di un processore, nonché la loro sintassi e semantica determinano il **linguaggio macchina** di quel processore
- Le istruzioni di un processore lavorano tipicamente su celle o insiemi di celle di memoria, ed eseguono operazioni elementari come somme, moltiplicazioni, copie di celle di memoria e simili
- In un tipico linguaggio macchina non esistono i concetti di variabile, tipo di dato, programmazione strutturata e così via

Linguaggio macchina 2/2

- I programmi in linguaggio macchina non sono in generale codificati in formato testo
- Ogni istruzione è invece individuata da una data sequenza di byte (ossia da un valore numerico)
- Anche gli operandi su cui lavora sono a loro volta individuati in qualche modo mediante delle sequenze di byte
- In definitiva, un programma in linguaggio macchina non è altro che una sequenza di byte, ossia di numeri

Linguaggio assembly

- Per poter rappresentare in modo leggibile per un essere umano un programma in linguaggio macchina si utilizza tipicamente un linguaggio chiamato *assembly*
- Dato un linguaggio assembly corrispondente ad un dato linguaggio macchina
 - Per ogni istruzione del linguaggio macchina esiste una corrispondente istruzione nel linguaggio assembly
 - In assembly tale istruzione non è più individuata da un numero, ma da una stringa
 - tipicamente un nome abbreviato che ricorda lo scopo dell'istruzione stessa

Registri

- Prima di vedere un semplicissimo frammento di codice assembly, ci occorre sapere cosa sono i registri di cui è tipicamente dotato un processore
 - Un registro è una speciale cella di memoria (tipicamente di dimensioni maggiori di un byte) interna al processore
- Uno degli scopi di tali registri è contenere gli operandi delle istruzioni aritmetiche
 - Infatti tali istruzioni tipicamente non possono utilizzare direttamente le celle della memoria principale dell'elaboratore come operandi
 - E' quindi spesso necessario copiare prima gli operandi all'interno dei registri

Esempio

- Il seguente frammento di codice scritto in un ipotetico linguaggio assembly semplificato (per un ipotetico processore reale) realizza le operazioni necessarie per calcolare il risultato dell'espressione aritmetica $2+3$ e memorizzarlo in un registro del processore stesso

```
LDA 3      # memorizza il valore 3
           # nel registro AX
ADD 2      # somma 2 al valore memorizzato nel
           # registro AX
```

Eseguibilità e portabilità

- Il linguaggio macchina è **il** linguaggio di **un** processore
- Il linguaggio macchina di un dato processore è direttamente eseguibile da un elaboratore basato su quel processore, senza alcuna intermediazione
- Processori differenti hanno linguaggi macchina differenti
- Pertanto, un programma scritto nel linguaggio macchina di un processore non è eseguibile su di un altro processore!
 - Si dice che non è portabile (ossia non può essere eseguito su piattaforme diverse)

Scelta linguaggio macchina

- Dato l'elaboratore su cui si intende eseguire un certo algoritmo, è una buona idea codificare direttamente nel linguaggio macchina del suo processore tale algoritmo?
 - **SI**, se
 - l'algoritmo è tale da poter essere implementato in tale linguaggio con sufficiente semplicità ed in modo molto efficiente, ed il problema necessita di tale elevata efficienza
 - la portabilità è un obiettivo secondario
 - **NO**, se non ci si trova nella precedente situazione
- Nel secondo caso è molto più efficace lavorare ad un più alto *livello di ASTRAZIONE*

Astrazione

- Ricordiamo che una definizione formale di processo di astrazione è
 - Attività di aggregazione di informazioni e dati, e di definizione di modelli concettuali che ne enucleano le proprietà rilevanti escludendo i dettagli inessenziali

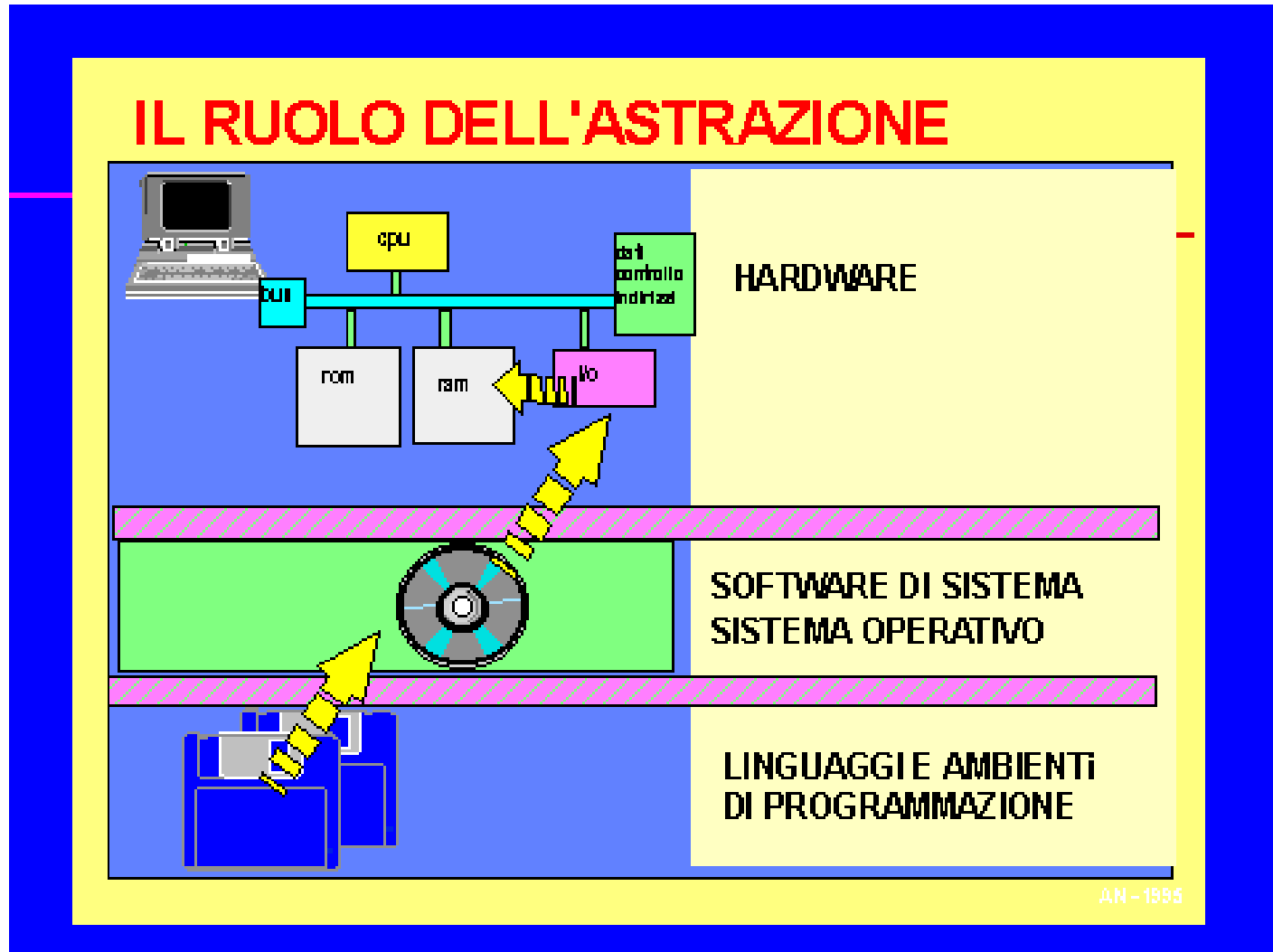
Esempio

- Ad esempio la definizione dei numeri `int` del linguaggio C/C++ è il risultato di un processo di astrazione delle celle di memoria in cui
 - Si definisce il modello concettuale di (sottoinsieme dei) numeri interi con le relative operazioni aritmetiche
 - Tale modello
 - enuclea la proprietà delle celle di memoria di memorizzare numeri interi con delle configurazioni di bit che possono essere opportunamente manipolate per realizzare anche le operazioni aritmetiche
 - Esclude dettagli inessenziali quali il modo esatto in cui un numero di tipo `int` è memorizzato su più celle di memoria o l'algoritmo con cui vanno manipolate le configurazioni di bit memorizzate nelle celle di memoria per effettuare le operazioni

Livello di astrazione

- Il livello di astrazione di un concetto/oggetto o di un linguaggio di programmazione è il livello a cui si trova tale concetto/oggetto o linguaggio di programmazione nell'ambito del processo di astrazione precedentemente definito
- Ossia quanto astrae dai dettagli dell'architettura di partenza

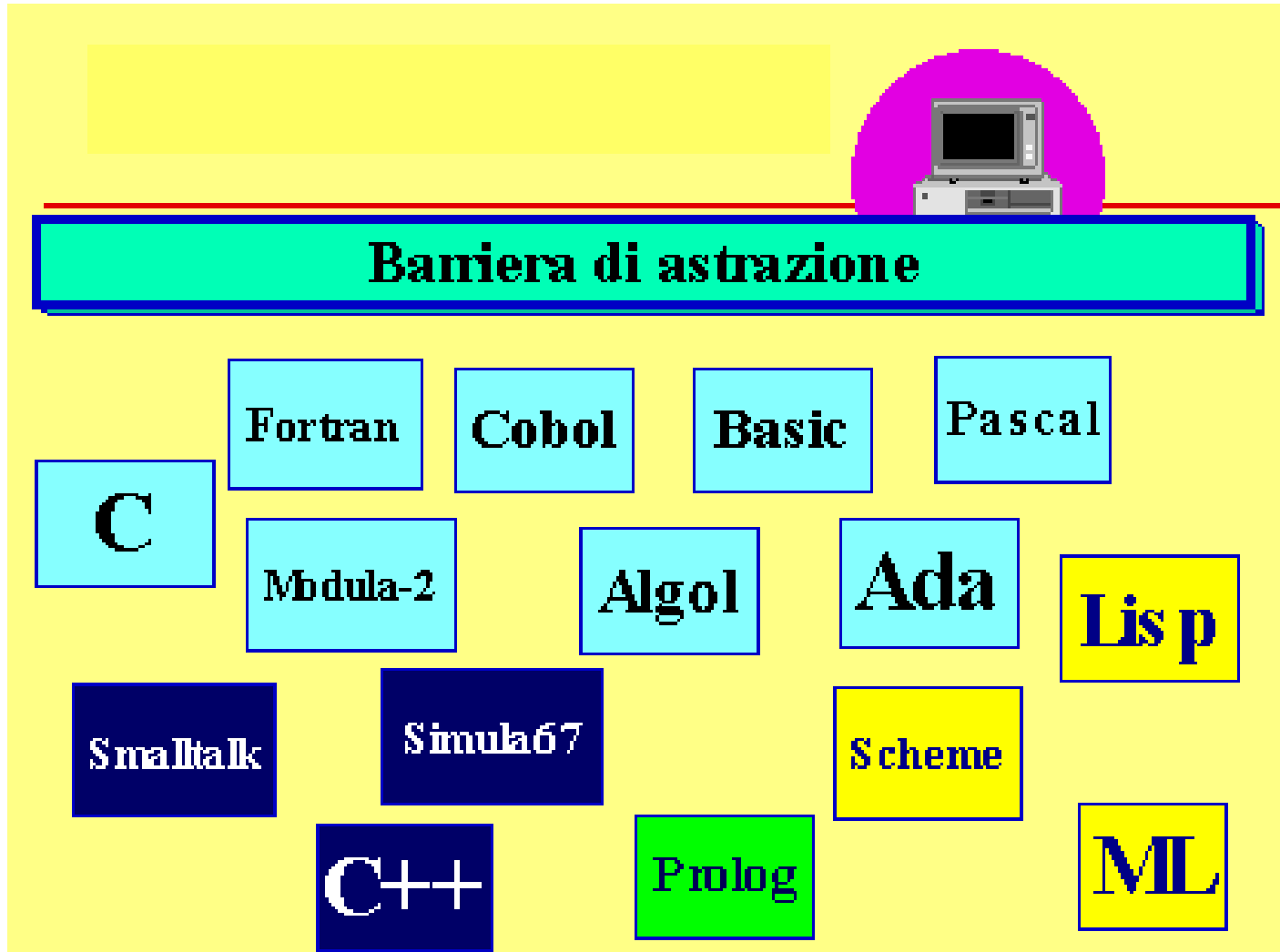
Astrazione di un elaboratore



Linguaggi di alto livello 1/2

- Si basano su un ipotetico elaboratore dotato di un processore le cui istruzioni non sono quelle di un tipico processore reale, ma quelle del linguaggio stesso
 - Ciò che abbiamo chiamato *macchina virtuale*
- Supportano concetti ed astrazioni
Esempio: variabili, tipo di dato, funzioni
- Promuovono metodologie per agevolare lo sviluppo del software da parte del programmatore
Esempio: programmazione strutturata e/o ad oggetti
- Hanno capacità espressive molto superiori rispetto a quelle del linguaggio macchina
Esempio: costrutti iterativi
- Esistono centinaia di linguaggi di programmazione di alto livello di astrazione! (anche se pochi sono ampiamente utilizzati)

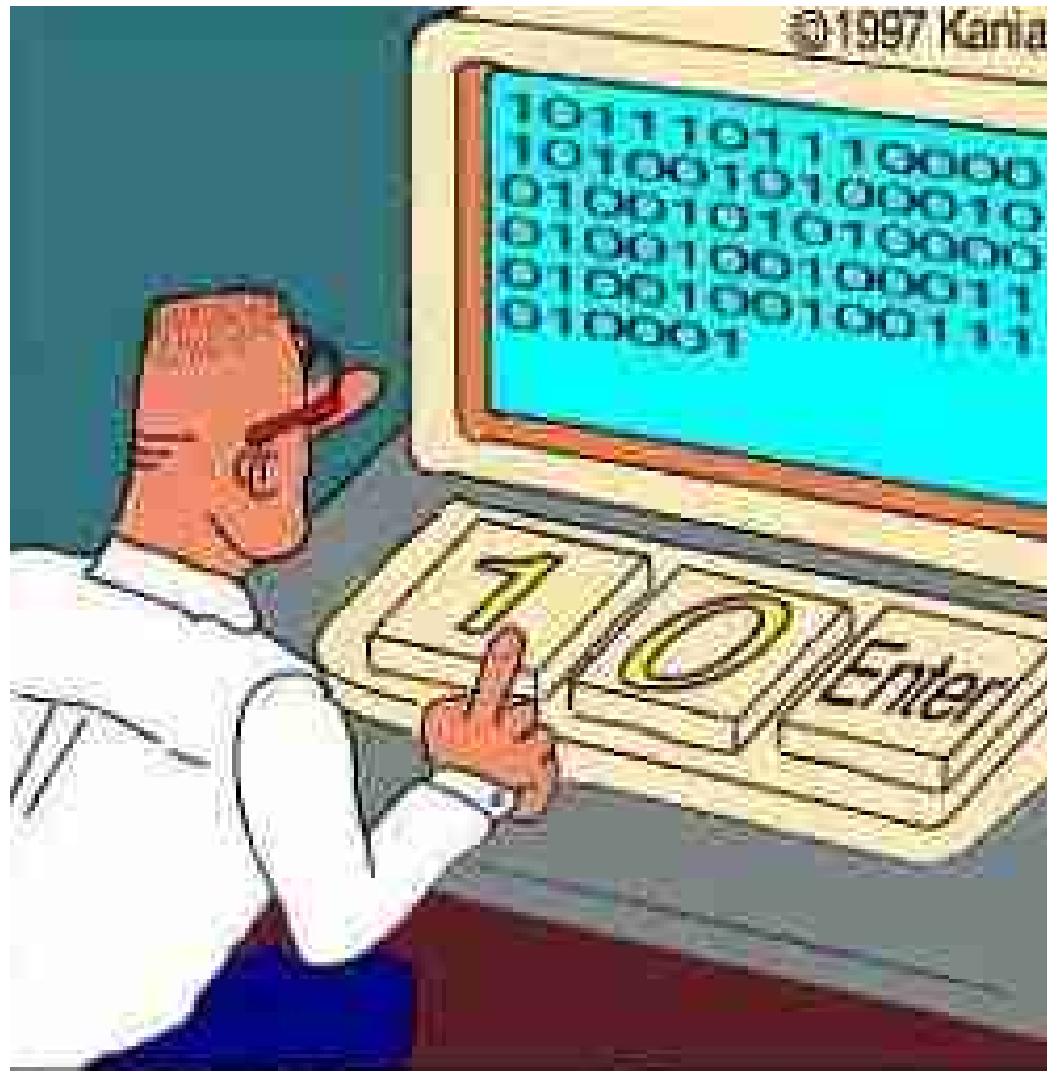
Linguaggi di alto livello 2/2



Nota conclusiva

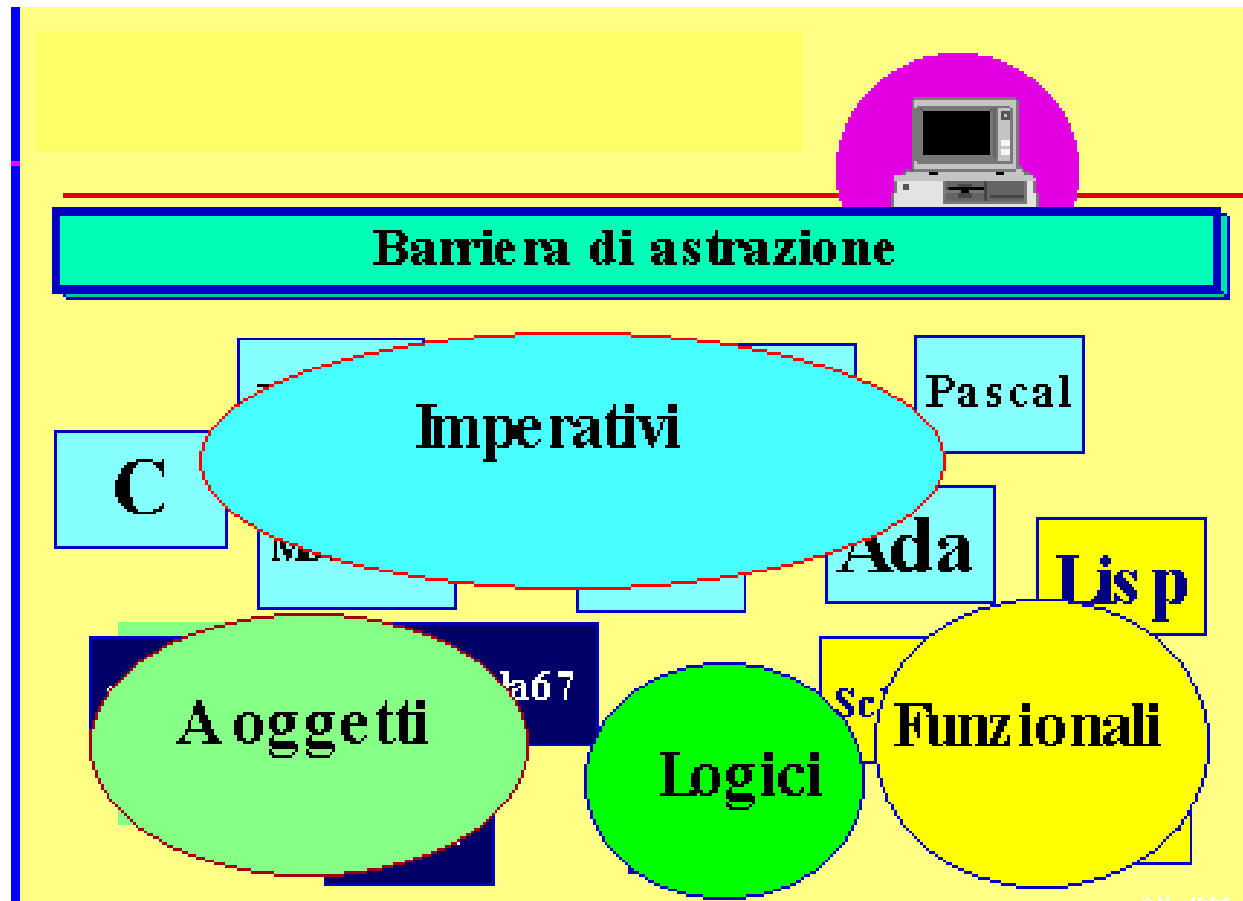
- Quindi, in generale un linguaggio di alto livello permette di codificare un algoritmo
 - con dei costrutti e delle strutture dati più vicine al dominio del problema
 - senza curarsi di quasi nessuno dei dettagli di basso livello del calcolatore su cui verrà eseguito il programma
 - ottenendo maggiore portabilità del linguaggio assembly
- Quindi il linguaggio assembly rimane la scelta migliore solo nei casi riportati precedentemente
- Ovviamente poi la scelta è obbligata se non si resiste al richiamo della seguente verità ...

I veri programmatori

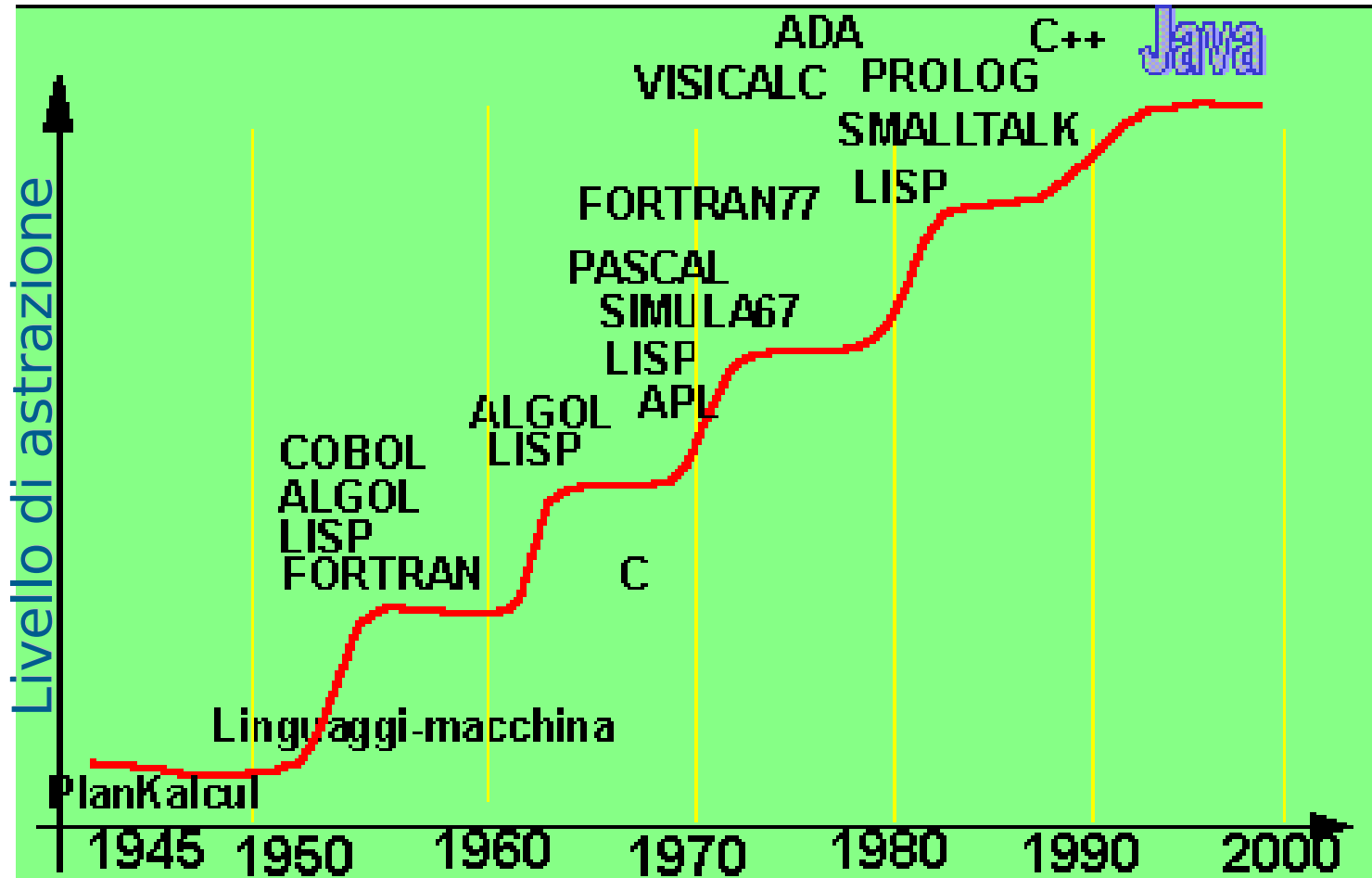


Real programmers code in binary.

Categorie



Evoluzione dei linguaggi



Perché esistono tanti linguaggi

- Uno dei motivi fondamentali è che tendono ad esistere diversi insiemi di linguaggi per ogni contesto applicativo, ad esempio:
 - Scientifico: Fortran
 - Gestionale: Cobol
 - Sistemi Operativi: C
 - Applicazioni (non di rete): C++, Java
 - Applicazioni di rete: Java

Problema

- Praticamente non esistono processori il cui linguaggio macchina raggiunga un livello di astrazione confrontabile con quello di un linguaggio di programmazione di alto livello
- Pertanto, non esistono processori in grado di eseguire più o meno direttamente programmi scritti in linguaggi ad alto livello

Traduzione

- Affinché un programma scritto in un qualsiasi linguaggio di programmazione ad alto livello sia eseguibile su un dato elaboratore,
 - occorre **tradurlo** dal linguaggio di programmazione originario al linguaggio macchina del processore montato su tale elaboratore
- Questa operazione viene normalmente svolta da speciali programmi, detti **traduttori**

Processo di traduzione

Programma originario

```
main()  
{ int A;  
  ...  
  A=A+1;  
  if ...
```

Programma tradotto

```
00100101  
  ...  
11001..  
1011100 ...
```

- I traduttori convertono il testo dei programmi scritti in un particolare linguaggio di programmazione, **programmi sorgenti**, nella corrispondente rappresentazione in linguaggio macchina, **programmi eseguibili**

Tipi di traduttori 1/2

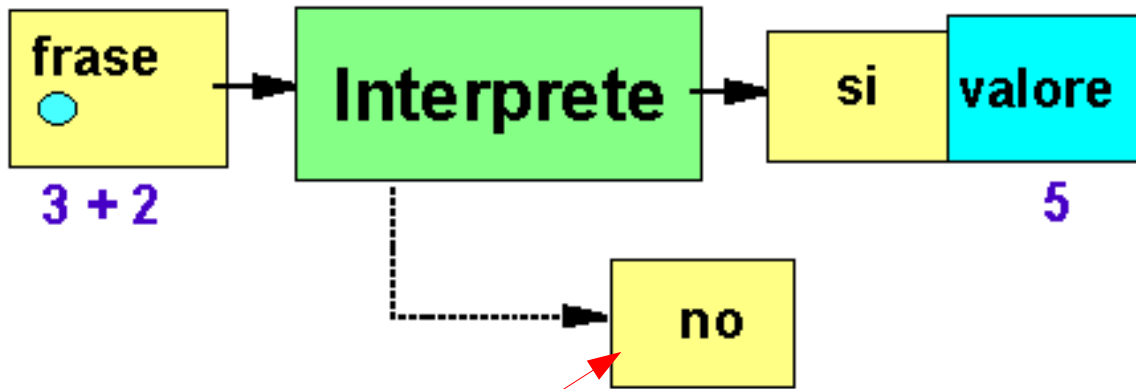
- Due categorie di traduttori:
 - **Compileri:** traducono l'intero programma, **senza eseguirlo!**, e producono in uscita il programma convertito in linguaggio macchina
 - Cosiddetto file eseguibile
 - **Interpreti:** traducono ed **eseguono immediatamente** ogni istruzione del programma sorgente, l'una dopo l'altra (tipicamente non viene prodotto alcun file)

Tipi di traduttori 2/2

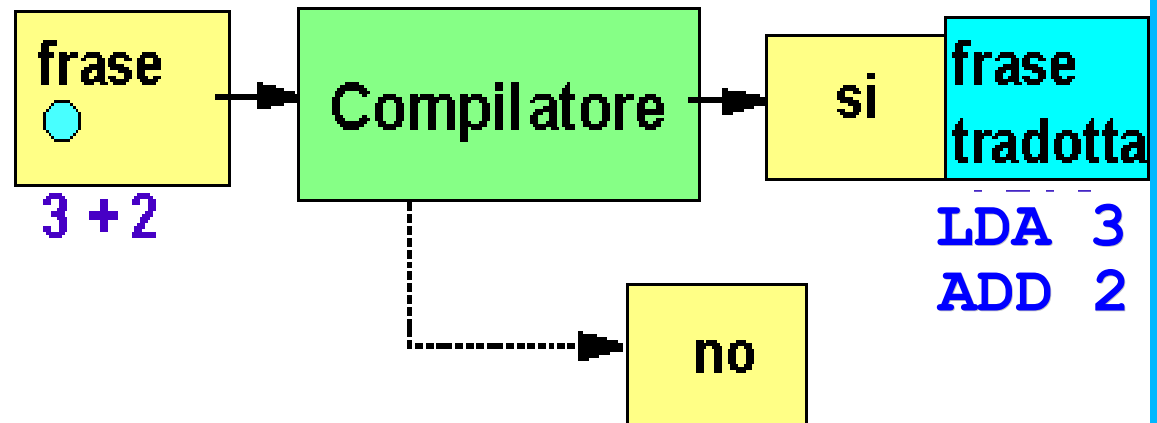
- Quindi,
 - **Nel caso del compilatore**, lo schema traduzione-esecuzione viene percorso una volta sola prima dell'esecuzione, e porta alla creazione del file eseguibile che sarà poi eseguito senza altri interventi
 - **Nel caso dell'interprete**, lo schema traduzione-esecuzione viene ripetuto tante volte quante sono le istruzioni del programma che saranno eseguite
 - Ad ogni attivazione dell'interprete su di una particolare istruzione segue l'esecuzione dell'istruzione stessa

Compilatore ed interprete 1/3

■ Riconoscimento (e valutazione)



■ Riconoscimento (e traduzione)



Frases non
corrette

Compilatore ed interprete 2/3

- Sebbene in linea di principio un qualsiasi linguaggio possa essere tradotto sia mediante compilatori sia mediante interpreti, nella pratica si tende verso una differenziazione già a livello di linguaggio:
 - Tipici linguaggi interpretati: Basic, Javascript, Perl, ...
 - Tipici linguaggi compilati: C/C++, Fortran, Pascal, ADA, ...
 - Java costituisce un caso particolare: si effettua prima una pre-compilazione per ottenere un codice intermedio tra l'alto livello ed il linguaggio macchina, poi l'interprete esegue tale codice intermedio

Compilatore ed interprete 3/3

- L'esecuzione di un programma compilato è tipicamente più veloce dell'esecuzione di un programma interpretato
 - Siccome la traduzione è effettuata una sola volta prima dell'esecuzione, di fatto poi si va ad eseguire direttamente il programma in linguaggio macchina
 - Grazie ai nuovi compilatori-interpreti i programmi Java sono eseguibili praticamente alla velocità di un programma compilato
- Un programma sorgente da interpretare è tipicamente più portabile di un programma da compilare

Header e file pre-compilati

- Gli header file contengono spesso solo delle dichiarazioni
- Come sappiamo, le dichiarazioni ci permettono di utilizzare degli oggetti
- Occorre però che poi questi oggetti siano definiti da qualche parte!
- In effetti le funzioni e gli oggetti di una libreria sono definiti in ancora altri file
 - Tali file sono tipicamente pre-compilati nel caso di linguaggi compilati
- In pratica, per fornire una certa libreria vengono forniti tanto gli header file, quanto dei file pre-compilati contenenti il codice macchina delle funzioni e degli oggetti messi a disposizione dalla libreria

Collegamento delle librerie 1/2

- Quindi, se in un programma includiamo correttamente gli header file di una certa libreria e ne usiamo le funzioni o gli oggetti, il tutto funziona se
 - Il compilatore **collega** il nostro programma ai file pre-compilati contenenti il codice macchina necessario
- E' per esempio quello che è accaduto automaticamente per la libreria di ingresso/uscita
 - Trattandosi di una libreria utilizzata molto spesso, i compilatori sono tipicamente pre-configurati per collegare il programma ai file pre-compilati di tale libreria

Collegamento delle librerie 2/2

- A volte però il compilatore può non essere già predisposto a collegare il nostro programma ai file pre-compilati di determinate librerie non utilizzate spesso
- In quel caso dobbiamo istruirlo noi, passando ad esempio opzioni aggiuntive
 - Lo abbiamo fatto quando abbiamo aggiunto l'opzione `-lm` per far collegare il nostro programma ai file pre-compilati della libreria matematica

Fasi della compilazione 1/3

- Tipicamente un compilatore ottiene un programma eseguibile da un programma sorgente attraverso varie fasi di compilazione

1) Preprocessing: il testo del programma sorgente viene modificato in base a delle semplici direttive.

Ne abbiamo già visto esempi in C/C++ con le direttive **#include** e **#define**

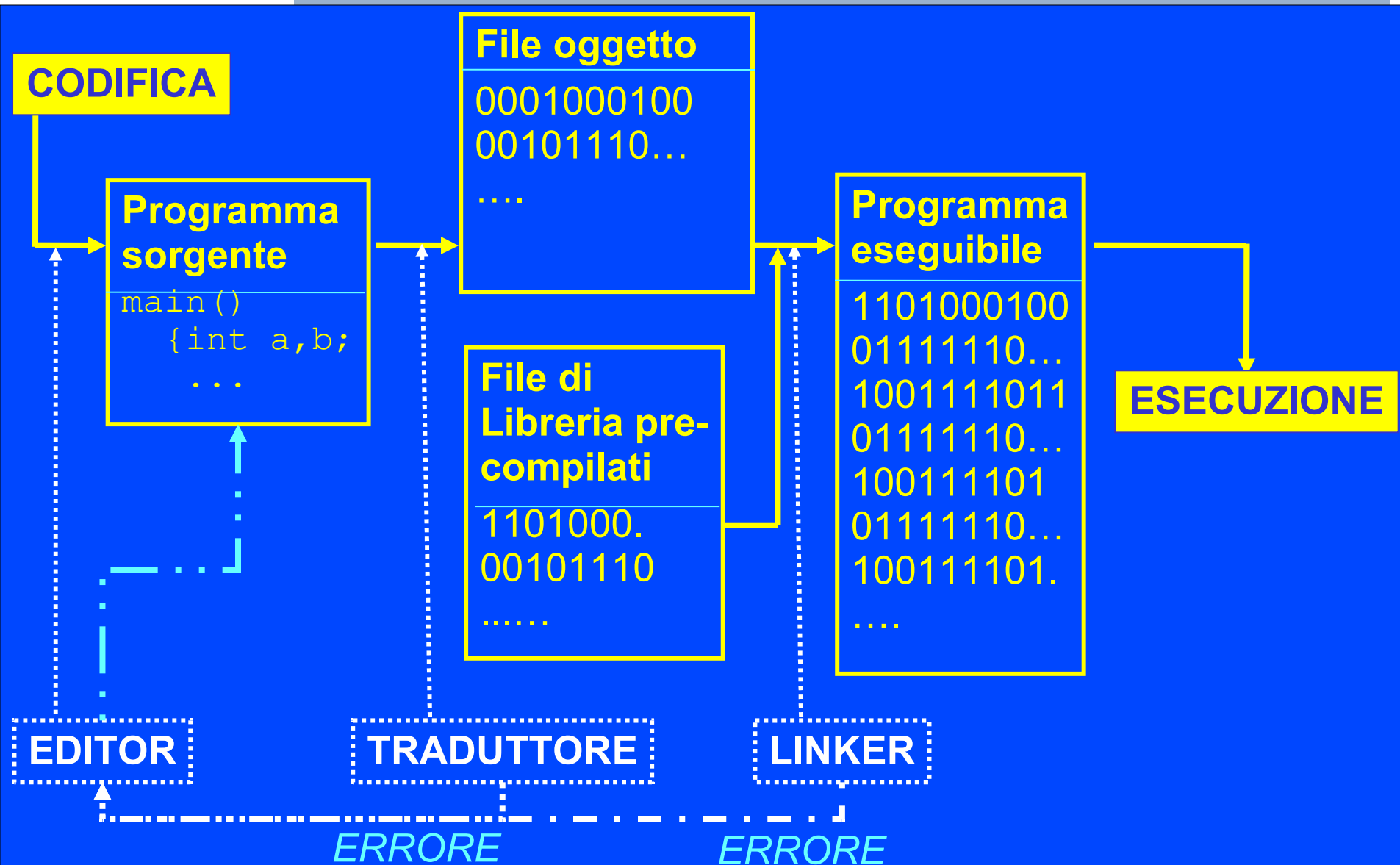
Fasi della compilazione 2/3

- 2) Traduzione** (spesso chiamata compilazione, come vedete c'è spesso confusione con i termini): genera un programma in linguaggio macchina a partire dal programma in linguaggio sorgente
- Il componente di un compilatore che realizza questa fase è tipicamente chiamato traduttore
 - Il programma generato nella fase di traduzione non è però tipicamente eseguibile, perché manca fondamentalmente il codice delle funzioni e degli oggetti non definiti nel programma stesso
 - Un programma in linguaggio macchina di questo tipo è tipicamente chiamato **file oggetto**

Fasi della compilazione 3/3

- 3) Collegamento:** si unisce il file oggetto con i file pre-compilati delle librerie (ed eventualmente con altri file oggetto nel caso di programmi sviluppati su più file sorgente)
- Il risultato è il programma (file) eseguibile
 - Il componente di un compilatore che realizza questa fase è tipicamente chiamato collegatore o linker

Schema riassuntivo (parziale)



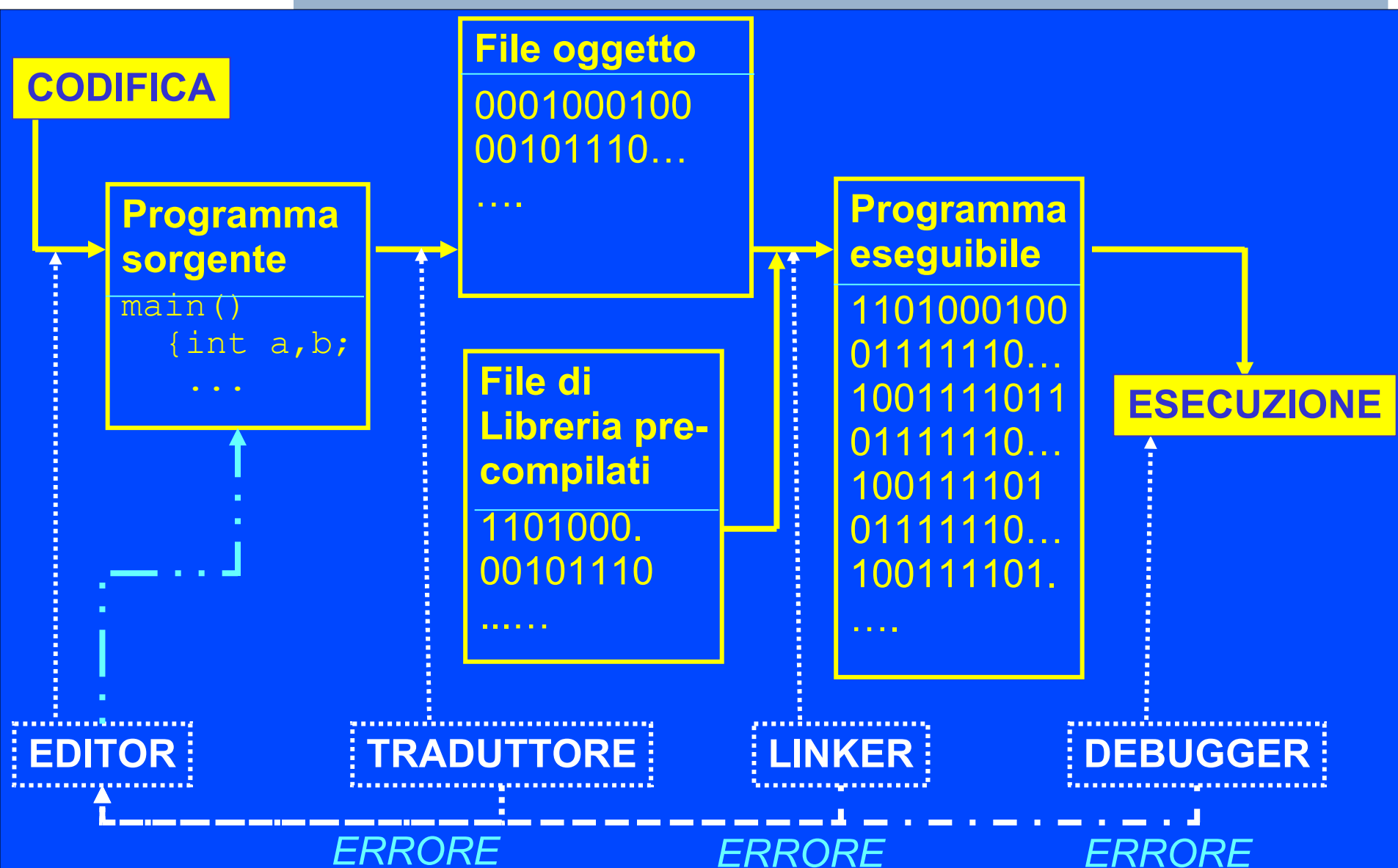
Fasi dello sviluppo

- Lo sviluppo di un programma passa attraverso varie *fasi*:
 - Progettazione
 - Scrittura (codifica)
 - Compilazione (per i linguaggi compilati)
 - Esecuzione
 - Collaudo (testing)
 - Ricerca ed eliminazione degli errori (debugging)
 - ...
- Si chiama tipicamente **ambiente di programmazione** (o di sviluppo) per un dato linguaggio o insieme di linguaggi, l'insieme degli strumenti (*tool*) di supporto alle varie fasi di sviluppo dei programmi scritti con tali linguaggi

Ambiente di programmazione

- **Editor:** serve per creare file di testo. In particolare, l'editor consente di scrivere il programma sorgente.
- **Traduttore** (spesso chiamato anche compilatore): opera la traduzione di un programma sorgente scritto in un linguaggio ad alto livello in un programma oggetto.
- **Linker** ("collegatore"): nel caso in cui la costruzione del programma eseguibile richieda l'unione di più moduli (compilati separatamente), provvede a collegarli per formare un unico file eseguibile.
 - Spesso traduttore e linker (e pre-processore) sono componenti di un dato compilatore
- **Interprete:** traduce ed esegue direttamente ciascuna istruzione del programma sorgente, istruzione per istruzione. È alternativo al compilatore.
- **Debugger:** consente di eseguire passo-passo un programma, controllando via via quel che succede, al fine di scoprire ed eliminare errori non rilevati in fase di compilazione.

Schema riassuntivo



Tipi di ambienti di sviluppo 1/2

- Si possono considerare fondamentalmente due tipologie di ambienti di sviluppo
 - 1) Ambienti dati dalla somma di componenti più o meno indipendenti
 - Un programma per l'editing, un programma per la compilazione, un programma per il debugging, ...
 - I sistemi operativi UNIX costituiscono spesso dei veri e propri ambienti di sviluppo di questo genere
 - sono dei sistemi in cui è possibile avere molta scelta per i singoli strumenti di sviluppo

Tipi di ambienti di sviluppo 2/2

2) Ambienti di sviluppo integrati

(Integrated Development Environment, IDE)

- Ambienti in cui i singoli strumenti sono integrati gli uni con gli altri e si dispone di un'unica interfaccia per gestire tutte le fasi (editing, compilazione, esecuzione, debugging, ...)

Confronto tra i tipi di ambiente

- Il vantaggio principale degli IDE è probabilmente la praticità e semplicità d'uso
 - Uso di una interfaccia comune (tipicamente grafica)
 - Possibilità di salvare, compilare ed eseguire premendo un solo bottone
 - Posizionamento automatico nei punti del programma in cui si trovano gli errori
- Vantaggi dei sistemi non IDE
 - Si distinguono perfettamente le varie fasi dello sviluppo (utilità fondamentale didattica)
 - Si ha maggiore indipendenza da uno specifico strumento (si può usare/cambiare lo strumento preferito per ciascuna fase dello sviluppo)