

ASTRAZIONE

Esistono linguaggi a vari livelli di astrazione

Linguaggio Macchina:

- implica la conoscenza dei metodi utilizzati per la rappresentazione delle informazioni

Linguaggio Macchina e Assembler (Assembly):

- implica la conoscenza dettagliata delle caratteristiche della macchina (registri, dimensioni dati, set di istruzioni)
- semplici algoritmi implicano la specifica di molte istruzioni

Linguaggi di Alto Livello:

- Il programmatore può astrarre dai dettagli legati all'architettura ed esprimere i propri algoritmi in modo simbolico



Sono indipendenti dalla macchina hardware sottostante
ASTRAZIONE

ASTRAZIONE

- **Linguaggio Macchina:**

```
0100 0000 0000 1000
0100 0000 0000 1001
0000 0000 0000 1000
```

Difficile leggere e capire un programma scritto in forma binaria

- **Linguaggio Assembler:**

```
... LOADA H
   LOADB Z
   ADD
   ...
```

Le istruzioni corrispondono univocamente a quelle macchina, ma vengono espresse tramite nomi simbolici (parole chiave)

- **Linguaggi di Alto Livello:**

```
main()
{ int A;
  scanf("%d", &A);
  if (A==0) {...}
...}
```

Sono indipendenti dalla macchina

ESECUZIONE

Per eseguire sulla macchina hardware un programma scritto in un *linguaggio di alto livello* è necessario tradurre il programma in *sequenze di istruzioni di basso livello*, direttamente eseguite dal processore, attraverso:

- interpretazione (ad es. BASIC)
- compilazione (ad es. C, FORTRAN, Pascal)

COME SVILUPPARE UN PROGRAMMA

Qualunque sia il linguaggio di programmazione scelto occorre:

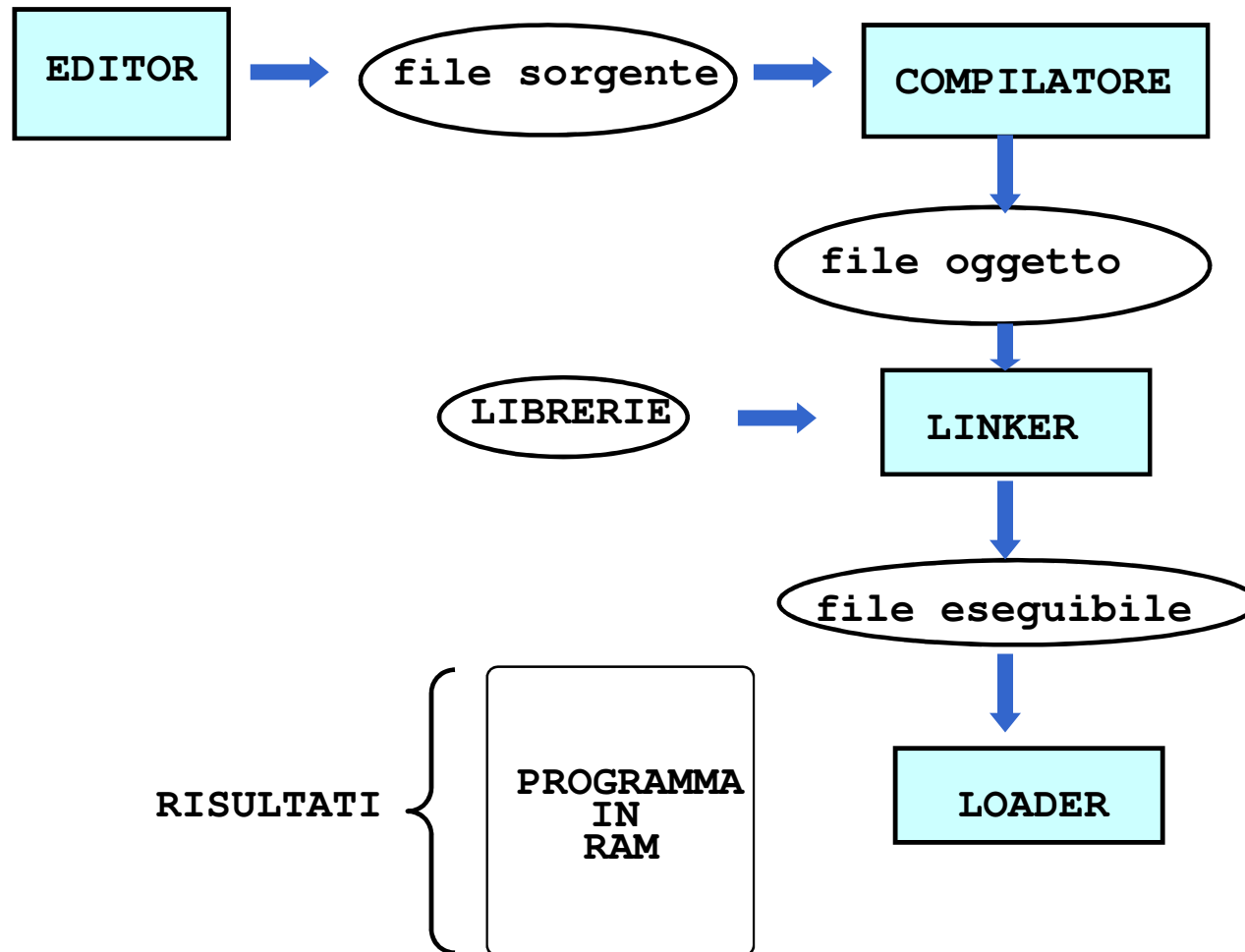
- *Scrivere il **testo del programma** e memorizzarlo su supporti di memoria permanenti (*fase di editing*)*
- Se il linguaggio è compilato:
 - *Compilare il programma, ossia utilizzare il compilatore che effettua una traduzione automatica del programma scritto in un linguaggio qualunque in un programma equivalente scritto in **linguaggio macchina***
 - *Eseguire il programma tradotto*
- Se il linguaggio è interpretato:
 - *Usare l'interprete per eseguire il programma*

COMPILATORI E INTERPRETI

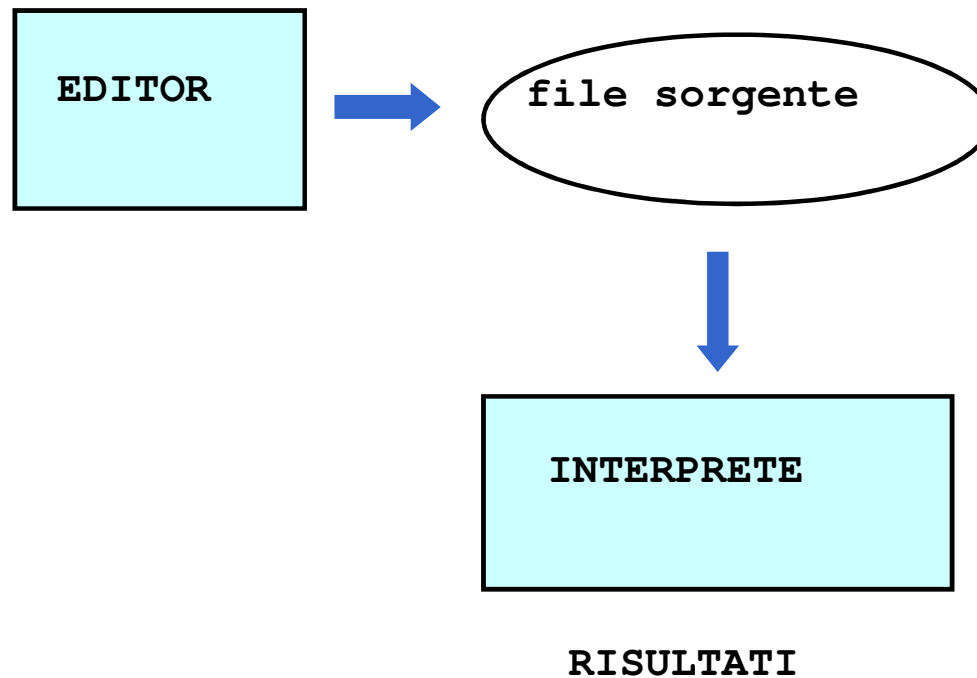
- I **compilatori** traducono automaticamente un programma dal linguaggio L a quello macchina (per un determinato elaboratore)
- Gli **interpreti** sono programmi capaci di eseguire direttamente un programma in linguaggio L istruzione per istruzione

I programmi compilati sono in generale *più efficienti* di quelli interpretati

APPROCCIO COMPILATO: SCHEMA



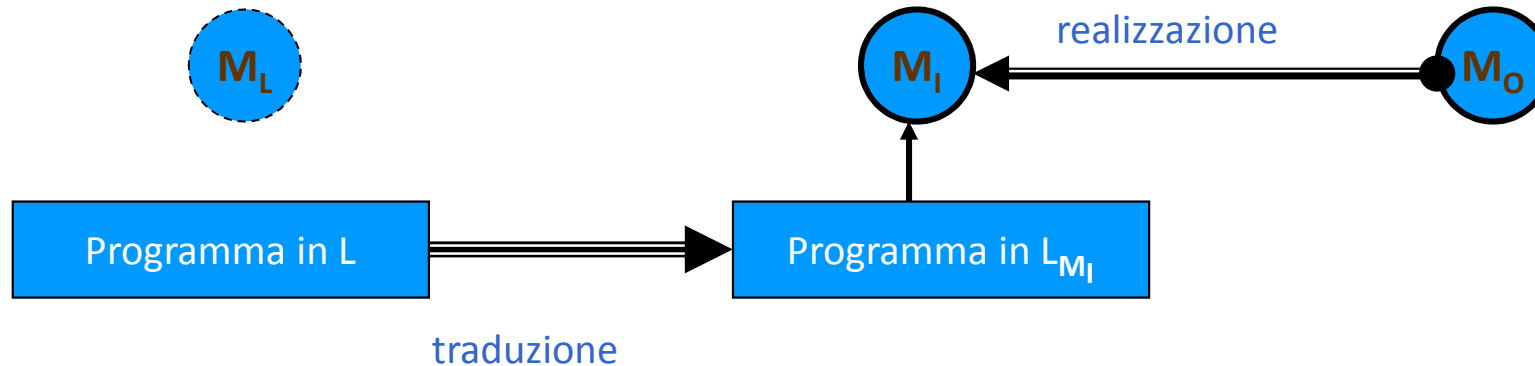
APPROCCIO INTERPRETATO: SCHEMA



Implementare un linguaggio di programmazione

- **L** linguaggio ad alto livello
- **M_L** macchina astratta di **L**
- **M_o** macchina ospite
- implementazione di **L 1: interprete** (puro)
 - **M_L** è realizzata su **M_o** in modo interpretativo
 - scarsa efficienza, soprattutto per colpa dell'interprete (ciclo di decodifica)
- implementazione di **L 2: compilatore** (puro)
 - i programmi di **L** sono tradotti in programmi funzionalmente equivalenti nel linguaggio macchina di **M_o**
 - i programmi tradotti sono eseguiti direttamente su **M_o**
 - **M_L** non viene realizzata
 - il problema è quello della dimensione del codice prodotto
- Esiste un approccio intermedio

La macchina intermedia



- **L** linguaggio ad alto livello
- **M_L** macchina astratta di **L**
- **M_I** macchina intermedia
- **L_{M_I}** linguaggio intermedio
- **M_O** macchina ospite
- traduzione dei programmi da **L** al linguaggio intermedio **L_{M_I}**+realizzazione della macchina intermedia **M_I** su **M_O**

Tre famiglie di implementazioni

□ interprete puro

- $M_L = M_I$
- interprete di L realizzato su M_O
- alcune implementazioni (vecchie!) di linguaggi logici e funzionali
 - LISP, PROLOG

□ compilatore

- macchina intermedia M_I realizzata per estensione sulla macchina ospite M_O (rts, nessun interprete)
 - C, C++, PASCAL

□ implementazione mista

- traduzione dei programmi da L a L_{M_I}
- i programmi L_{M_I} sono interpretati su M_O
 - Java
 - i “compilatori” per linguaggi funzionali e logici (LISP, PROLOG, ML)
 - alcune (vecchie!) implementazioni di Pascal (Pcode)

COMPILATORI: MODELLO

La costruzione di un compilatore per un particolare linguaggio di programmazione è complessa

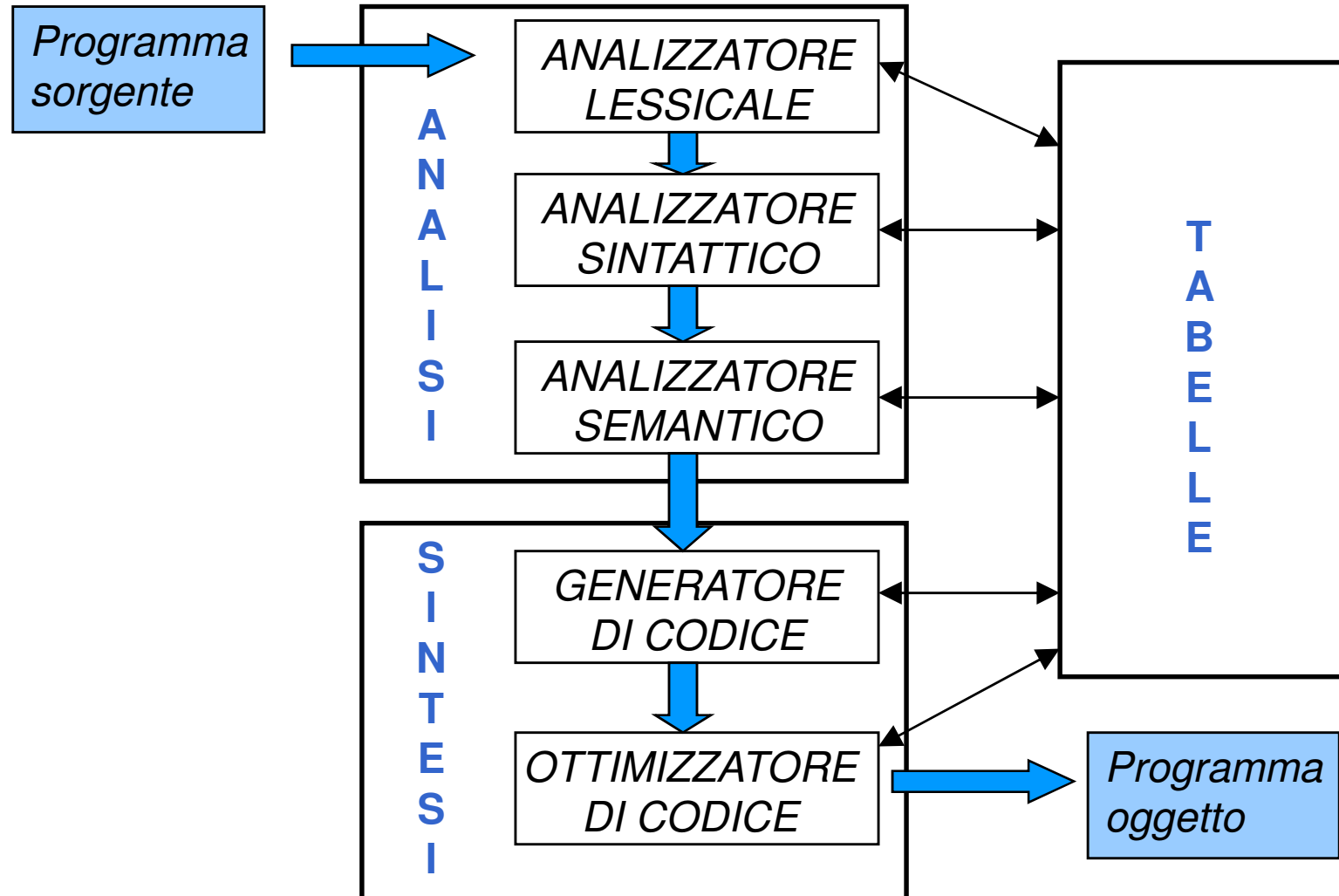
- La complessità dipende dal linguaggio sorgente

Compilatore: traduce il programma sorgente in programma oggetto

Due compiti:

- **ANALISI** del programma sorgente
- **SINTESI** del programma oggetto

COMPILATORI: MODELLO



ANALISI

Il compilatore nel corso dell' analisi del programma sorgente verifica la correttezza sintattica e semantica del programma:

- **ANALISI LESSICALE** verifica che i simboli utilizzati siano legali cioè appartengano all' alfabeto
- **ANALISI SINTATTICA** verifica che le regole grammaticali siano rispettate => albero sintattico
- **ANALISI SEMANTICA** verifica i vincoli imposti dal contesto

SINTESI

Generatore di codice: trasla la forma intermedia in linguaggio assembler o macchina

Prima della generazione di codice:

- ALLOCAZIONE DELLA MEMORIA
- ALLOCAZIONE DEI REGISTRI

Eventuale passo ulteriore di **ottimizzazione del codice**

LINGUAGGI DI PROGRAMMAZIONE

Il “potere espressivo” di un linguaggio è caratterizzato da:

- *quali **tipi di dati** consente di rappresentare (direttamente o tramite definizione dell'utente)*
- *quali **istruzioni di controllo** mette a disposizione (quali operazioni e in quale ordine di esecuzione)*

PROGRAMMA = DATI + CONTROLLO

IL LINGUAGGIO C: Storia

- 1969 Ken Thompson scrive la prima versione del Sistema Operativo Unix in Assembler. Si occupa anche di pensare ad un linguaggio di più alto livello (B).
- definito nel 1972 da Dennis Ritchie (AT&T Bell Labs) a partire dal linguaggio B e utilizzato per riscrivere quasi totalmente Unix.
- prima definizione precisa: Kernigham & Ritchie (1978), libro: “The C Programming Language”.

IL LINGUAGGIO C: lo Standard

- Utilizzato su diverse architetture con diversi dialetti, nasce la necessità di uno standard per usarlo in modo portabile da parte dell'ANSI (America National Standard Institute - 1983)
- 1989 nasce lo Standard ANSI C, C89, , C90...
- 1999 nuova versione estesa C99
- 2011 C11 con altre estensioni e maggiore compatibilità con C++
- C99 e C11 più ricchi ma non supportati da tutti i compilatori.

IL LINGUAGGIO C

CARATTERISTICHE

- *linguaggio* **sequenziale, imperativo, strutturato a blocchi, basato su espressioni**
- *usabile anche come linguaggio di sistema*
 - *adatto a software di base, sistemi operativi, compilatori, ecc.*
- *portabile, efficiente (compilato), sintetico*
 - *ma a volte poco leggibile...*

IL LINGUAGGIO C

Basato su pochi concetti elementari

- *dati (tipi primitivi, tipi di dato)*
- *espressioni*
- *dichiarazioni / definizioni*
- *funzioni*
- *istruzioni / blocchi*

ESEMPIO: un semplice programma

Codifica in linguaggio C dell' algoritmo che converte gradi Celsius in Fahrenheit

```
int main() {  
    float c, f; /* Celsius e Fahrenheit */  
    printf("Inserisci la temperatura da convertire");  
    scanf("%f", &c);  
    f = 32 + c * 9/5;  
    printf("Temperatura Fahrenheit %f", f);  
}
```

STRUTTURA DI UN PROGRAMMA C

In prima battuta, la struttura di un programma C è definita nel modo seguente:

```
<programma> ::=  
{ <unità-di-traduzione> }  
<main>  
{ <unità-di-traduzione> }
```

Intuitivamente un programma in C è definito da tre parti:

- una o più unità di traduzione
- il programma vero e proprio (main)
- una o più unità di traduzione

STRUTTURA DI UN PROGRAMMA C

La parte <main> è l'unica obbligatoria, definita come segue:

```
<main> ::=  
    int main()  
    { [<dichiarazioni-e-definizioni>]  
      [<sequenza-istruzioni>]  
    }
```

Intuitivamente il main è definito dalla parola chiave `main()` e racchiuso tra parentesi graffe al cui interno troviamo

- dichiarazioni e definizioni
 - una sequenza di istruzioni
- } *opzionali []*

STRUTTURA DI UN PROGRAMMA C

- **<dichiarazioni-e-definizioni>**

introducono i nomi *di costanti, variabili, tipi definiti dall'utente*

- **<sequenza-istruzioni>**

*sequenza di frasi del linguaggio
ognuna delle quali è un'istruzione*

main () è una particolare unità di **traduzione** (*una funzione*)

ALFABETO BASE DEL LINGUAGGIO C

- ***Set di caratteri ammessi in un programma dipende dall'implementazione; solitamente ASCII + estensioni.***

Compaiono almeno 96 simboli:

- 26 caratteri minuscoli e maiuscoli dell'alfabeto inglese
- 10 Cifre decimali
- 29 caratteri grafici
- 5 caratteri di spaziatura

LESSICO DEL C

- ***Regole lessicali:*** set di regole per definire parole sull'alfabeto del linguaggio
- ***Categorie lessicali:***
 - *Parole Chiave (Keywords)*
 - *Costanti*
 - *Identificatori*
 - *Commenti*

Identificatori

- *sequenze di caratteri tali che*

<Identificatore> ::=

<Lettera> { <Lettera> | <Cifra> }

Intuitivamente un identificatore è una sequenza (di lunghezza maggiore o uguale a 1) di lettere e cifre che **inizia obbligatoriamente con una lettera**

Non può essere una parola chiave;

Non ci sono limiti alla lunghezza, ma sono determinati dall'implementazione.

Parole Chiave in C

Un nome non deve coincidere con una parola riservata, né con il nome di una funzione di libreria, a meno che non si desideri creare una propria versione della funzione

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

COMMENTI

*Sequenze di caratteri racchiuse fra i delimitatori **/*** e ***/** ignorate dal Compilatore (e rimosse dal preprocessore)*

```
<Commento> ::= /* <frase> */  
<frase> ::= { <parola> }  
<parola> ::= { <carattere> }
```

*I commenti **non** possono essere innestati*

In C++ (ma di solito accettata anche in C)

// sono un commento

(vale fino alla fine della riga)

VARIABILI

- **Una variabile è un'astrazione di una cella di memoria**
- **Formalmente, è un simbolo associato a un indirizzo fisico (L-value)...**

<i>simbolo</i>	<i>indirizzo</i>
X	1328

Perciò, L-value di x è 1328 (fisso e immutabile!)

VARIABILI

... che denota un valore (R-value)



... e R-value di x è attualmente 4 (può cambiare)

DEFINIZIONE DI VARIABILE

*Una variabile utilizzata in un programma
deve essere definita*

*La **definizione** è composta da*

- **nome della variabile (identificatore)**
- **tipo dei valori (R-value)** che possono essere denotati alla variabile

*e implica **allocazione di memoria**
necessaria a mantenere R-value denotato*

DEFINIZIONE DI VARIABILE: ESEMPI

Definizione di una variabile:

<tipo> <identificatore>;

int x; / x deve denotare un valore intero */*

float y; / y deve denotare un valore reale */*

char ch; / ch deve denotare un carattere */*

INIZIALIZZAZIONE DI UNA VARIABILE

- *Contestualmente alla definizione è possibile specificare un valore iniziale per una variabile (altrimenti il suo contenuto è non definito)*
- *Inizializzazione di una variabile:*

<tipo> <identificatore> = <espr> ;

Esempio

```
int x = 32;
```

```
double speed = 124.6;
```

VARIABILI & ESPRESSIONI

Una variabile

- *può comparire in una espressione*
- *può assumere un valore dato dalla valutazione di un'espressione*

```
double speed = 124.6;  
double time = 71.6;  
double km = speed * time;
```

CARATTERISTICHE DELLE VARIABILI

campo d'azione (scope): è la parte di programma in cui la variabile è nota e può essere manipolata

- *in C, Pascal*: determinabile **staticamente**
- *in LISP*: determinabile **dinamicamente**

tipo: specifica la **classe di valori** che la variabile può assumere (e quindi gli **operatori applicabili**)

CARATTERISTICHE DELLE VARIABILI

tempo di vita: è l'intervallo di tempo in cui rimane valida l'associazione simbolo/indirizzo (L-value)

- in FORTRAN: allocazione statica
- in C, Pascal: anche allocazione dinamica

valore: è rappresentato (secondo la codifica adottata) nell'area di memoria associata alla variabile

ESEMPIO: un semplice programma

Problema:

“Data una temperatura espressa in gradi Celsius, calcolare il corrispondente valore espresso in gradi Fahrenheit”

Approccio:

- *si parte dal **problema** e dalle **proprietà** note sul **dominio dei dati***

ESEMPIO: un semplice programma

Specifica della soluzione:

$$c * 9/5 = f - 32$$

oppure

$$c = (f - 32) * 5/9$$

$$f = 32 + c * 9/5$$

ESEMPIO: un semplice programma

Algoritmo corrispondente:

- *Dato c*
- *calcolare f sfruttando la relazione*
$$f = 32 + c * 9/5$$

solo a questo punto

***si codifica l'algoritmo nel linguaggio
scelto***

ESEMPIO: un semplice programma

```
int main() {  
    float c=18; /* Celsius */  
    float f = 32 + c * 9/5;  
}
```

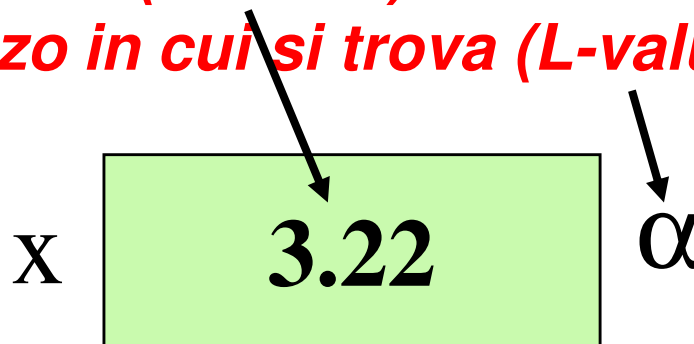


NOTA: per ora abbiamo a disposizione solo il modo per inizializzare le variabili. Mancano, ad esempio, la possibilità di modificare una variabile, costrutti per l'input/output...

VARIABILI NEI LINGUAGGI IMPERATIVI

Una variabile in un linguaggio imperativo

- non è solo un sinonimo per un dato come in matematica
- **è un'astrazione della cella di memoria**
- **associata a due diverse informazioni:**
 - **il contenuto (R-value)**
 - **l'indirizzo in cui si trova (L-value)**



ESPRESSIONI

- *Il C è un linguaggio basato su **espressioni***
- *Una **espressione** è una notazione che **denota un valore** mediante un processo di **valutazione***
- *Una espressione può essere semplice o composta (tramite aggregazione di altre espressioni)*

ESPRESSIONI CON EFFETTI COLLATERALI

- *Le espressioni che contengono variabili, oltre a denotare un valore, possono a volte comportare **effetti collaterali** sulle variabili coinvolte*
- *Un **effetto collaterale** è una modifica del valore della variabile (R-value) causato da particolari operatori:*
 - *operatore di **assegnamento***
 - *operatori di **incremento e decremento***

ASSEGNAMENTO

Ad una variabile può essere assegnato un valore nel corso del programma e non solo all'atto della inizializzazione

L'assegnamento è l'astrazione della modifica distruttiva del contenuto della cella di memoria denotata dalla variabile

□ **Assegnamento di una variabile: SINTASSI**

<identificatore> = <espr>

A sinistra non ci può essere una costante o un'espressione composta:

X+1=10 errore!

int x = 32;



x

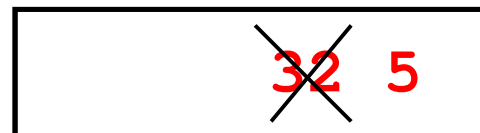


...

x = 5;



x



ASSEGNAMENTO

- *L'assegnamento è un particolare tipo di espressione*

come tale denota comunque un valore

*con un effetto collaterale: quello di **cambiare il valore della variabile***

- *Esempi di espressioni di assegnamento:*

$$j = 0$$

$$k = j + 1$$

- *Se k valeva 2, l'espressione $k = j + 1$*

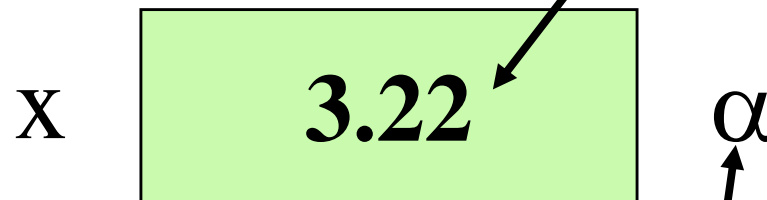
- *denota il valore 1 (risultato della valutazione dell'espressione)*
- *e cambia il valore di k , che d'ora in poi vale 1 (non più 2)*

L'assegnamento è distruttivo

ASSEGNAIMENTO & VARIABILI

Una variabile in una espressione di assegnamento:

- *è interpretata come il suo R-value, se compare a destra del simbolo =*



- *è interpretata come il suo L-value, se compare a sinistra del simbolo =*

ASSEGNAIMENTO & VARIABILI

Se x valeva 2, l'espressione

$$x = x + 1$$

denota il valore 3

e cambia in 3 il valore di x

- *il simbolo x **a destra** dell'operatore = denota il valore attuale (R-value) di x, cioè 2*
- *il simbolo x **a sinistra** dell'operatore = denota la cella di memoria associata a x (L-value), a cui viene assegnato il valore dell'espressione di destra (3)*
- *l'espressione nel suo complesso denota il **valore della variabile** dopo la modifica, cioè 3*

Assegnamento multiplo

- Un assegnamento è considerato un'espressione con un risultato. E' quindi possibile eseguire assegnamenti multipli su una sola riga di codice.
- `int x, y = 1, z=0; z=x=y=2;`
- Associativo a destra, equivale a: `z = (x = (y=2)) ;`
cioè:
- `y=2;`
- `x=2;`
- `z=2;`
- Nota: `z=2=y;` Errato!

OPERATORI DI ASSEGNAZIONE COMPATTI

Il C introduce una forma particolare di assegnamento che ingloba anche un' operazione:

<identificatore> OP= <espressione>

è "equivalente" a

*<identificatore> = <identificatore> OP
< espressione>*

*dove OP indica un operatore (ad esempio:
+, -, *, /, %,*

Esempi

k += j equivale a k = k + j

*k *= a + b equivale a k = k * (a+b)*