

# Programmazione (imperativa)

Corso di Laurea in Informatica

Roberto Cordone

DI - Università degli Studi di Milano



Lezioni:	Lunedì 12.00 - 13.00 e 14.00 - 16.00	Mercoledì 14.00 - 17.00
Laboratorio:	Giovedì 12.00 - 13.00 e 14.00 - 17.00	
Ricevimento:	su appuntamento	
Tel.:	02 503 16235	
E-mail:	roberto.cordone@unimi.it	
Web page:	<a href="http://homes.di.unimi.it/~cordone/courses/2015-prog/2015-prog.html">http://homes.di.unimi.it/~cordone/courses/2015-prog/2015-prog.html</a>	

**Blocco** è un insieme di istruzioni

- consecutive, con un punto di inizio e un punto di fine
- da eseguire nelle stesse condizioni
- che puntano a un solo scopo specifico
- che lavorano su un insieme coerente di dati
- che producono un insieme coerente di risultati

Un blocco può contenere altri blocchi annidati al suo interno

Programmi e funzioni sono blocchi, ma non sono i soli esempi

In C, **blocco** è un insieme di istruzioni chiuso fra parentesi graffe

Qualsiasi blocco non banale ha bisogno di conservare risultati parziali:  
**variabile** è un **oggetto che conserva un risultato parziale**

Ogni blocco ha una parte dichiarativa e una parte esecutiva

La parte dichiarativa indica tipo e nome delle variabili del blocco

- `char b;`

Una variabile è **locale al blocco** che la dichiara, cioè  
**può essere usata solo nel blocco stesso** (o in blocchi interni)

⇒ si può riusare lo stesso nome solo in blocchi diversi

Una variabile ha un'esistenza limitata:

- comincia a esistere con la sua dichiarazione
- la prima istruzione che la cita deve assegnarle un valore (**inizializzazione**)
- le istruzioni successive possono
  - usarne il valore come dato per fare altri calcoli
  - modificarne il valore, sostituendolo con il risultato di altri calcoli
- cessa di esistere alla fine della funzione che la dichiara

Non inizializzare una variabile favorisce gli errori

L'operatore di assegnamento = assegna a una variabile un valore

*variabile = valore ;*

```
char b;  
int l;  
float f;
```

```
b = '*';  
l = 17;  
f = 13.68;
```

Si possono combinare dichiarazione e inizializzazione (`int i = 1;`)  
ma questo è sconsigliabile se l'uso della variabile è lontano

Variabile e valore devono essere dello stesso tipo (in teoria...)

Il valore può essere descritto da una espressione

**Espressione** è una sequenza di simboli a cui è associato un valore:

- 1 una costante esplicita o simbolica:    '\*'    17    1.15    LARGHEZZA
- 2 una variabile:                                b        11        12
- 3 una chiamata a funzione:                CalcolaSomma(11,12)
- 4 un'espressione composta:                11 + LARGHEZZA
  - un operatore (simbolo che rappresenta un'operazione)
  - uno o più operandi, che sono a loro volta espressioni

Poiché anche gli operandi sono espressioni, **le espressioni hanno una struttura gerarchica** potenzialmente molto complessa

Il **valore di un'espressione** è

- ① per una costante: il suo **valore**
- ② per una variabile: il suo **valore corrente**, cioè l'ultimo valore assegnatole
- ③ per una chiamata a funzione: il **valore restituito**, cioè il risultato della funzione
- ④ per un'espressione composta: il **valore ottenuto eseguendo l'operazione indicata dall'operatore sui valori degli operandi**

Le espressioni composte vanno quindi **valutate ricorsivamente**:

- prima si valutano gli operandi
- poi si esegue l'operatore sui valori degli operandi

Il C offre moltissimi operatori:

- **aritmetici**: +, -, \*, /, %, ...
- **relazionali**: >, <, ==, ...
- **logici**: &&, ||, ...
- ...

**Arietà** (o **rango**) di un operatore è il **numero dei suoi operandi**

- gli **operatori unari** hanno **un solo operando**
- gli **operatori binari** hanno **due operandi**
- ...



# Regole di precedenza

Se un'espressione ha più operatori, servono **regole di precedenza** per definire l'ordine con il quale eseguire le operazioni associate

Ogni operatore ha una **priorità**; si considerano

- prima gli operatori a priorità superiore
- poi gli operatori a priorità inferiore

Tuttavia la priorità non ordina completamente gli operatori

Se un'espressione ha più operatori della stessa priorità, l'**associatività** determina se eseguire le operazioni associate

- da sinistra a destra
- da destra a sinistra

In questo modo si ottiene un ordinamento completo

Possono essere

- unari: **-** **cambia il segno** di un'espressione intera o reale
- binari:
  - **+** e **-** sono le consuete **somma** e **differenza**
  - **\*** è il **prodotto**
  - **/** è la **divisione** (**esatta** per numeri reali, **troncata** per numeri interi:  $7 / 3$  vale 2)
  - **%** è il **resto** della divisione intera:  $7 \% 3$  vale 1

Ammettono **operandi a valori interi o reali**

(in linea di principio, o tutti interi o tutti reali)

Producono espressioni con **valore dello stesso tipo**

La priorità e associatività sono quelle consuete

# Operatori relazionali

Dati due operandi, indicano se il primo è

- maggiore ( $>$ ), maggiore o uguale ( $>=$ )  
minore ( $<$ ), minore o uguale ( $<=$ )
- uguale ( $==$ ), diverso ( $!=$ )

rispetto al secondo

Questi operatori

- sono binari
- hanno **operandi di tipo intero o reale o carattere**  
(in linea di principio, tutti interi, tutti reali o tutti caratteri)
- producono espressioni con **valore di tipo logico** (vero o falso)

In ordine di priorità decrescente

- unario: **!** rappresenta la **negazione (NOT)**
- binari:
  - **&&** rappresenta la **congiunzione (AND)**
  - **||** rappresenta la **disgiunzione (OR)**

Questi operatori

- hanno **operandi di tipo logico**
- producono espressioni con **valore di tipo logico**

**Gli operatori aritmetici precedono quelli relazionali**  
**Gli operatori relazionali precedono quelli logici**

# Uso delle parentesi

Si possono usare le parentesi tonde ( ), anche su più livelli

- per rendere più chiaro l'ordine degli operatori
- per modificare l'ordine imposto dalle regole di precedenza

Ogni espressione fra parentesi è un operando

$(i + j) * k$	$i + (j * k)$
valutare $(i + j)$	valutare $i$
valutare $k$	valutare $(j * k)$
applicare $*$	applicare $+$

Se le parentesi non bastano, conviene

- **introdurre variabili ausiliarie:** `int l;`
- **scomporre le espressioni in sottoespressioni:** `(i+j) k`
- **assegnare il valore delle sottoespressioni alle variabili:** `l = i + j;`
- **sostituire le variabili alle sottoespressioni:** `l * k`

In C non esistono valori logici; convenzionalmente si considera

- falso il valore nullo (0)
- vero ogni valore intero diverso da zero

Gli operatori logici restituiscono il valore 1

dopo l'istruzione `i = (3 < 6)`; la variabile `i` vale 1

Questo rende lecito scrivere

- espressioni dal significato misterioso

```
int b1 = 1;           (cioè vero)
int b2 = 0;           (cioè falso)
( (b1 || b2) < b1 ) + 7   vale 7
```

- espressioni dal significato ingannevole

```
int i = 3;
(10 < i < 6)   vale vero
```

**Macro** è una **costante simbolica** definita con una direttiva **#define**  
`#define LUNGHEZZA 17`

**Il valore di una macro può essere un'espressione composta**

Siccome la direttiva indica solo una meccanica sostituzione di testo, conviene **racchiudere l'espressione fra parentesi per garantire la priorità**

Se si scrive ... il precompilatore legge e lo trasforma in Quindi <i>i</i> vale	<code>#define SOMMA 3 + 2</code> <code>i = 5 * SOMMA;</code> <code>i = 5 * 3 + 2;</code> 17
------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------

Se si scrive ... il precompilatore legge e lo trasforma in Quindi <i>i</i> vale	<code>#define SOMMA (3 + 2)</code> <code>i = 5 * SOMMA;</code> <code>i = 5 * (3 + 2);</code> 25
------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------