5. Tipi di dato ed espressioni

Andrea Marongiu

(andrea.marongiu@unimore.it)

Paolo Valente

UNIMORE

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Memoria di un programma

- Definiamo **memoria** di un programma in esecuzione, o *processo*, il contenitore (logico) in cui sono memorizzati tutti i dati del programma (ed altre informazioni che vedremo in seguito) durante la sua esecuzione
- Nei programmi C/C++ la memoria di un programma ha la stessa identica struttura della memoria del calcolatore vista precedentemente: è una sequenza contigua di **celle** (**locazioni di memoria**) che costituiscono l'unità minima di memorizzazione
- Le celle, tutte della stessa dimensione, contengono un *byte* ciascuna

Dimensione byte

- L'esatta dimensione che deve avere un *byte* non è specificata nello standard del linguaggio C/C++, e, come abbiamo visto, teoricamente può variare da una macchina all'altra
  - Lo standard specifica solo che un byte **deve** essere grande abbastanza da contenere un oggetto di tipo **char**
  - Vedremo in seguito cosa è un oggetto di tipo **char**, per ora ci basta sapere che è utilizzato principalmente per memorizzare caratteri

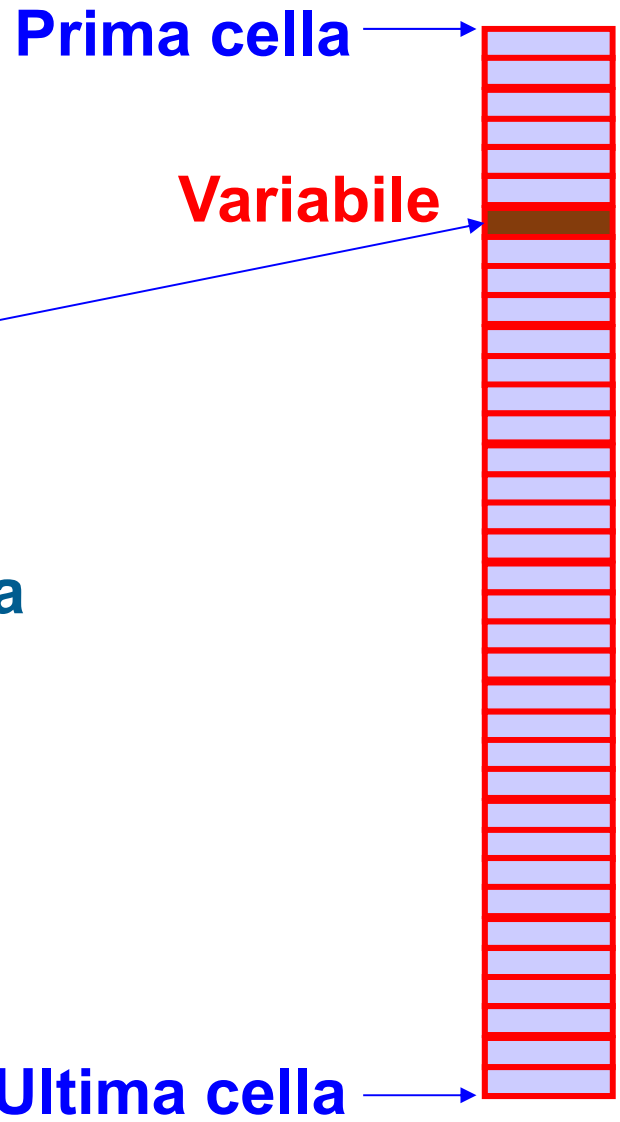
Dalle celle ai dati

- In C/C++ si possono memorizzare delle informazioni più complesse dei semplici numeri interi rappresentabili con una singola cella di memoria
- Si possono memorizzare i dati all'interno di «contenitori»
  - che il programma astrae come **variabili**

Variabile, valore, memoria

- Una **variabile** è un'astrazione di cella di memoria
  - E' caratterizzata da un **valore**
  - E' memorizzata in una **sequenza di celle contigue**
    - Consideriamo per esempio, come oggetto, un numero naturale maggiore di 255
    - Come abbiamo visto, così come si può rappresentare ogni numero naturale da 0 a 255 con una determinata configurazione di 8 bit, si può rappresentare un valore naturale maggiore di 255 su *N* celle consecutive, con una determinata configurazione dei risultanti 8**N* bit

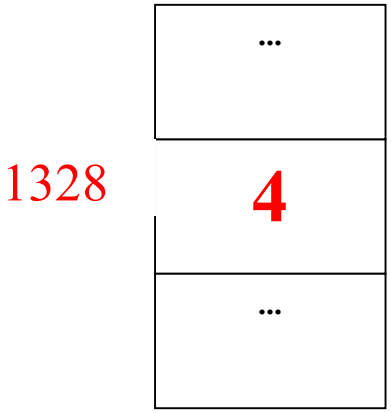
Variabile in memoria, indirizzo



Indirizzo della variabile in memoria: indirizzo della **prima cella della sequenza** di celle occupate dall'oggetto

Indirizzo, valore e tipo 1/2

- Una variabile è caratterizzata da
  - un ***indirizzo***
    - Ad esempio 1328, il che vuol dire che la variabile (il dato) si trova in memoria a partire dalla cella di indirizzo 1328



  - un ***valore***
    - In questo semplice esempio la variabile è di tipo numerico, occupa una sola cella e la configurazione di bit della cella rappresenta il valore 4

Indirizzo, valore e tipo 2/2

- un **tipo (di dato)**
  - Specifica i valori possibili per l'oggetto e le operazioni che si possono effettuare sull'oggetto

Tipo di dato

- **Tipo di un dato (variabile)**
- Insieme di valori che la variabile può assumere ed insieme di operazioni che si possono effettuare su quell'oggetto

- Quali tipi di dato esistono in C/C++?
  - Partiamo dai tipi di dato primitivi

Tipi di dato primitivi

- Quattro tipi di dato primitivi

Nome tipo	*Categoria di dati che rappresenta*
int	sottoinsieme dei numeri interi
float	sottoinsieme dei numeri reali
double	sottoinsieme dei numeri reali con maggiore precisione rispetto al tipo **float**
char	caratteri
bool	booleani (vero/falso, solo C++)

Per ora vedremo più in dettaglio il solo tipo **int**

Tipo **int**

- Il tipo **int** è diverso dal tipo INTERO inteso in senso matematico, dove l'insieme infinito degli interi Z è dato da
- {…,-2,-1,0,+1,+2,…}
- Ovvero il tipo **int** ha un insieme di valori limitato:
  - *L'insieme esatto dei valori possibili dipende dalla macchina*
  - Normalmente il compilatore è configurato in maniera tale che gli oggeti di tipo **int** siano memorizzati in una PAROLA DI MACCHINA, che tipicamente è lunga 2, 4 o 8 byte, ossia 16, 32 o 64 bit
  - Se la macchina ha parole a 16 bit:

    $[-2^{15}, 2^{15}-1]$ ovvero [-32768, +32767]
  - Se la macchina ha parole a 32 bit:

    $[-2^{31}, 2^{31}-1]$ ovvero [-2147483648, +2147483647]

  ...

Operazioni aritmetiche **int**

- Al tipo **int** sono applicabili i seguenti operatori:
-

+ Addizione

- Sottrazione

***** Moltiplicazione

/ Divisione intera (diverso dalla divisione reale!)
Es., 10/3 = 3

% Modulo (resto della divisione intera)
Dati tre numeri naturali *divid*, *divis* e *ris*, dove ris=divid/divis (divisione intera), il resto è il numero naturale *res* tale che
divid = *ris* * *divis* + *res*
Es., 10%3 = 1
5%3 = 2

Esempio

```
int v;          // definizione variabile v
v = 4;          // assegna il valore 4
                // alla variabile v
v = 2 * 3;      // assegna il valore 6
                // alla variabile v
```

In seguito, vedremo in dettaglio tutti i tipi di *espressioni* che si possono scrivere

Espressioni letterali

- Le espressioni letterali denotano **valori costanti**
- Sono spesso chiamate semplicemente *letterali* o *costanti senza nome*
- Le possibili espressioni letterali utilizzabili in
- C/C++ sono
  - numeri interi
  - numeri reali
  - costanti carattere
  - costanti stringa
- Vedremo le ultime tre categorie più avanti

Numeri interi

In quanto invece ai numeri interi, ecco alcuni ovvi esempi dei letterali utilizzabili in un programma C/C++:

```
6          12          700
```

Costanti con nome

- Una definizione di una *costante con nome* associa permanentemente un oggetto di valore costante ad un identificatore
- La definizione è identica a quella di una variabile, a parte
  - Aggiunta della parola chiave **`const`** all'inizio
  - Obbligo di inizializzazione

Esempi:

```
const int N = 100;
const int L ;    // errato: manca
                 // inizializzazione
```

Costanti e variabili

- Una costante è un'astrazione **simbolica** di un valore: si da cioè un nome ad un valore
- E' una associazione identificatore-valore che **non cambia mai** durante l'esecuzione
- **Non si può** quindi **assegnare un nuovo valore ad una costante** mediante una istruzione di assegnamento

- Invece, nel caso di una **variabile**
  - L'associazione identificatore-indirizzo non cambia mai durante l'esecuzione, ma può cambiare l'*associazione identificatore-valore*
  - Uno stesso identificatore può denotare valori differenti in momenti diversi dell'esecuzione del programma

Struttura programmi

- In questo insegnamento vedremo solo programmi sviluppati su di un unico file sorgente
  - Vedrete lo sviluppo di un programma su più file nel corso di **Programmazione II**
- Nelle prossime slide iniziamo a vedere la struttura semplificata di un programma
- Come primo passo, per motivare la presenza delle cosiddette *direttive* in un programma, partiamo dal menzionare il *pre-processore*

Pre-processore

- Prima della compilazione vera e propria, il file sorgente viene manipolato dal cosiddetto **pre-processore**, il cui compito è effettuare delle modifiche o delle aggiunte al testo originario
- La nuova versione del programma viene memorizzata in un **file temporaneo**, ed è questo il vero file che viene passato al compilatore
  - Il file temporaneo è poi automaticamente distrutto alla fine della compilazione
- Vedremo in seguito cosa fa il pre-processore in dettaglio, quello che ci basta sapere per ora è che il pre-processore viene pilotato dal programmatore mediante le cosiddette **direttive** inserite nel file sorgente

Dichiarazioni e definizioni

- Nelle prossime slide metteremo in evidenza un tipo di istruzioni chiamate **dichiarazioni**
  - Una dichiarazione è una istruzione in cui si introduce un nuovo identificatore
- Le definizioni sono casi particolari di dichiarazioni
  - Sono dichiarazioni la cui esecuzione provoca l'allocazione di spazio in memoria
  - In particolare, la definizione di una variabile o di una costante con nome provoca l'allocazione di spazio in memoria per la variabile o costante che viene definita

Struttura programma C

```
#include <stdio.h>
main()
{
        <dichiarazione>
        <dichiarazione>
        ...
        <dichiarazione>
        <istruzione diversa da dichiarazione>
        <istruzione diversa da dichiarazione>
        ...
        <istruzione diversa da dichiarazione>
}
```

Direttive per il pre-processore

Obbligatorio nei vecchi standard: prima tutte le dichiarazioni, poi qualsiasi altro tipo di istruzione

Struttura programma C++

```
#include <iostream>
using namespace std ;
main()
{
      <istruzione qualsiasi>
      <istruzione qualsiasi>
      ...
      <istruzione qualsiasi>
}
```

Direttive per il pre-processore

Diversamente dal C, in qualsiasi standard del C++ si possono mescolare tutti i tipi di istruzioni

Funzione *main*

- *main()* è una funzione speciale con tre caratteristiche:
  - deve essere sempre presente
  - la prima istruzione della funzione *main()* è la prima istruzione del programma che sarà eseguita, indipendentemente da dove si trova la funzione *main()* all'interno del file sorgente
  - quando termina l'esecuzione del *main()*, ossia dopo dopo l'esecuzione dell'ultima istruzione contenuta nella funzione *main()*, termina l'intero programma

- Come si è visto, in C la funzione *main()* contiene due sezioni
  - Parte dichiarativa
  - Parte esecutiva vera e propria

Ordine di esecuzione

- In che ordine vengono eseguite le istruzioni?
- Si definisce **sequenza** o **concatenazione** una sequenza di istruzioni scritte l'una di seguito all'altra all'interno di un programma
- Le istruzioni/dichiarazioni di una sequenza sono **eseguite l'una dopo l'altra**

ESEMPIO

```
int N ;           // prima si esegue la definizione
N = 3 ;           // poi l'assegnamento
cout<<N<<endl;    // infine la stampa
```

Soluzione problemi

- Data la centralità di questo aspetto per la vostra professionalità
  - Nella prossima slide rivediamo la procedura corretta per passare da un problema ad una soluzione valida

Sviluppo di una soluzione

- Un buon ordine con cui arrivare a risolvere, mediante un programma, un problema nuovo di cui non si conosce la soluzione è il seguente:

1) Riflettere sul problema finché non si è sicuri di aver capito a sufficienza tutti gli aspetti e le implicazioni
2) Cercare di farsi venire un'idea che sembri buona per risolvere il problema (o almeno per partire)
3) Provare a definire l'algoritmo e controllarlo per capire se è corretto (eventualmente modificarlo)
4) Quando si è sicuri dell'algoritmo, partire con la codifica
5) Collaudare il programma per verificare che faccia veramente quello che deve

Istruzione di assegnamento

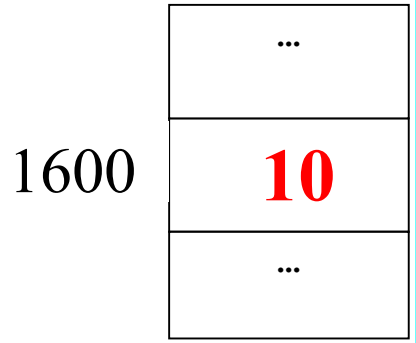
- Espressione di assegnamento:
-
- *nome_variabile* **=** <*espressione*>
-
- Istruzione di assegnamento:
-
- <*espressione di assegnamento*> **;**
  - E' cioè una espressione di assegnamento **seguita da un ;**
  - Viene utilizzata per assegnare ad una variabile (non ad una costante!) il valore di un’espressione

Assegnamento e memoria

Esempio

```
int  N=10;
```

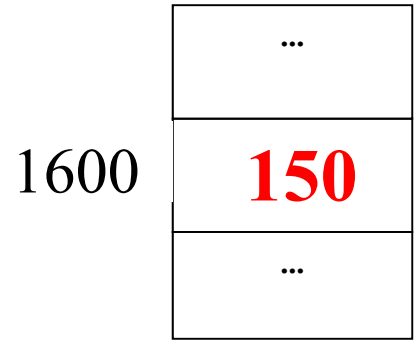
simbolo	*indirizzo*
N	**1600**



L'esecuzione di una **definizione** provoca l'allocazione di uno spazio in memoria pari a quello necessario a contenere un dato del tipo specificato

```
N = 150;
```

simbolo	*indirizzo*
N	**1600**



L'esecuzione di un **assegnamento** provoca l'inserimento nello spazio relativo alla variabile del valore indicato a destra del simbolo **=**

Assegnamento e memoria

- Quale informazione bisogna avere per poter modificare il valore di una variabile in memoria?

- Bisogna sapere dove si trova in memoria!
- Occorre sapere cioè il suo **indirizzo**
  - Ossia l'indirizzo della prima delle celle consecutive in cui è memorizzata la variabile

lvalue e rvalue

- Come si effettua quindi l'assegnamento?
- Consideriamo, per esempio, il precedente assegnamento:
- **`N = 150;`**
- Viene preso l'indirizzo della variabile individuata dall'identificatore a sinistra dell'assegnamento (l'identificatore è **`N`** nel nostro esempio)
  - tale indirizzo è detto **lvalue** (left value)
- Viene calcolato il valore dell'espressione che compare a destra (150 nell'esempio) ed assegnato all'oggetto memorizzato all'indirizzo (lvalue) ottenuto nel passo precedente
  - tale valore è detto **rvalue** (right value)

Ordine di esecuzione

- L'esecuzione di un'istruzione di assegnamento comporta **prima** la valutazione di tutta l'espressione a destra dell'assegnamento.
- Esempi:
- ```
  int c, d;
  ```
- ```
  c = 2;
  ```
- ```
  d = (c+5)/3 - c;
  ```
- ```
  d = (d+c)/2;
  ```
- Solo **dopo** si inserisce il valore risultante (*rvalue*) nella spazio di memoria dedicato alla variabile

Risultato assegnamento 1/2

- Come tutte le espressioni, anche l'espressione di assegnamento ha un proprio valore
- In particolare ha per valore *l'indirizzo della variabile a cui si è assegnato il nuovo valore* (quindi l'*lvalue*)
- Esempio: l'espressione
- **`a = 3`**
- ha per valore l'indirizzo di **`a`**
- Uno dei modi in cui si può sfruttare tale indirizzo è per effettuare *assegnamenti multipli*, ad esempio:
- ```
  int c, d;
  ```
- ```
  c = d = 2;
  ```
  - L'effetto della seconda istruzione, che, come si vedrà meglio in seguito, è equivalente a
  - **`c = (d = 2) ;`**
  - è il seguente:

Risultato assegnamento 2/2

- L'espressione **d = 2** produce come valore l'indirizzo della variabile **d**
- L'espressione **c =** … si aspetta a destra un valore da assegnare a **c**
  - Siccome si ritrova invece l'indirizzo di una variabile, tale indirizzo viene utilizzato per accedere al (nuovo) valore della variabile **d** (ossia 2), ed utilizzarlo per assegnare il nuovo valore a **c**
- In definitiva dopo l'istruzione
- **c = (d = 2) ;**
- sia **c** che **d** hanno il valore 2

Tipo booleano

- Disponibile in C++, ma non in C
- Nome del tipo: **`bool`**
- Valori possibili: vero (**true**), falso (**false**)
  - Identificati dai due letterali booleani:
  - **`true`** e **`false`**
- Esempio di definizione:
-
- ```
  bool u, v = true ;        // la seconda variabile
                            // è inizializzata a
                            // vero
  ```
- Operazioni possibili: ...
  ```

Operatori logici: sintassi

operatore logico	*numero argomenti*	**sintassi (posizione)**	**esempi**
not logico (negazione)	*uno (unario)*	**!** (prefisso)	`bool b, a = !true ;` `b = !a ;`

and logico (congiunzione)	*due (binario)*	**&&** (infisso)	`bool b, a, c ;`
			`c = a && b ;`
			`b = true && a ;`

or logico (disgiunzione)	*due (binario)*	**||** (infisso)	`bool b, a, c ;`
			`c = a || b ;`
			`b = true || a ;`

Che valori ritornano questi operatori?
La loro semantica è definita dalle cosiddette *tabelle di verità*

Tabella di verità

AND			*Ris.*	OR			*Ris.*	NOT	*Ris.*
V	&&	V	V	V	||	V	V	!V	F
V	&&	F	F	V	||	F	V	!F	V
F	&&	V	F	F	||	V	V		
F	&&	F	F	F	||	F	F		

Tipo booleano e tipi numerici

- Se un oggetto di tipo booleano è usato dove è atteso un valore numerico
  - **`true`** è convertito a 1
  - **`false`** è convertito a 0
- Viceversa, se un oggetto di tipo numerabile è utilizzato dove è atteso un booleano
  - ogni valore diverso da 0 è convertito a **`true`**
  - il valore 0 è convertito a **`false`**

Tipo booleano e linguaggio C

- In C, non esistendo il tipo **`bool`**, gli operatori logici
  - operano su interi
    - il valore 0 viene considerato falso
    - ogni valore diverso da 0 viene considerato vero
  - e restituiscono un intero:
    - il risultato è 0 o 1
- Esempi di espressioni con operatori logici (che in C++ ritornerebbero **true** o **false**)
-
- **`5 && 7`** **`0 || 33`** **`!5`**

Booleani e valori interi in C++

- Anche in C++ si possono utilizzare gli interi dove sono attesi dei booleani, esattamente come in C
- Tali valori sono convertiti nel modo seguente:
  - il valore 0 viene convertito a **false**
  - ogni valore diverso da 0 viene convertito a **true**
- Ovviamente utilizzare i booleani al posto di *sovraccaricare* il significato degli interi
  - Rende i programmi molto più chiari
  - E' esattamente il motivo per cui sono stati introdotti i booleani

Operatori di confronto 1/2

== Operatore di confronto di uguaglianza
(il simbolo = denota invece l'operazione di assegnamento!)

!= Operatore di confronto di diversità

> Operatore di confronto di maggiore stretto
< Operatore di confronto di minore stretto
>= Operatore di confronto di maggiore-uguale
<= Operatore di confronto di minore-uguale

- Restituiscono un valore di tipo **booleano**:
  `true` oppure **`false`**

Operatori di confronto 2/2

- Gli operatori di confronto si possono applicare sia agli oggetti di tipo **int** che agli oggeti di tipo **bool**
  - E, come vedremo in seguito, anche ad altri tipi di oggetti

Espressioni

- Costrutto sintattico formato da letterali, identificatori, operatori, parentesi tonde, ...
- Operatori
  - Moltiplicativi: * / %
  - Additivi: + -
  - Traslazione: << >> (Programmazione II)
  - Relazione (confronto): < > <= >=
  - Eguaglianza (confronto): == !=
  - Logici: ! && || (ce ne sono anche altri)
  - Assegnamento: = += -= *= /=
- Abbiamo già visto quasi tutti questi operatori parlando del tipo **int** e del tipo **bool**

Domanda

- Data una variabile booleana **x**, che differenza c'è tra i valori delle espressioni per ciascuna delle seguenti coppie?
-

```
x == true          x

x == false     !x
```

Risposta

- **Nessuna**
- Un programmatore utilizza sempre la forma sintatticamente e concettualmente più semplice per una data espressione
- Quindi nelle precedenti coppie di espressioni sono da preferire:
-
- **x**
-
- **!x**
-

Altri operatori

- Assegnamento abbreviato: +=, -=, *=, /=, …
- **`a += b ;`** ↔ **`a = a + b ;`**
- Incremento e decremento: **`++ --`**
  - Prefisso: prima si effettua l'incremento/decremento, poi si usa la variabile. Restituisce un **lvalue** (l'indirizzo della variabile incrementata)
  -
  - **`int a = 3; cout<<++a; // stampa 4`**
  - **`(++a) = 4; // valido, cosa assegna ad a?`**
  - Postfisso: prima si usa il valore della variabile, poi si effettua l'incremento/decremento. Restituisce un **rvalue**
  -
  - **`int a = 3; cout<<a++; // stampa 3`**
  - **`(a++) = 7; // ERRORE !!!`**

Tipi di espressioni

- Un'espressione si definisce
  - aritmetica: produce un risultato di tipo aritmetico
  - logica: produce un risultato di tipo booleano
- Esempi:

Espressioni aritmetiche

```
2 + 3

( 2 + 3 ) * 5
```

```
4 > 2

true || (2 > 5)
```

Espressioni logiche

Proprietà degli operatori 1/2

- **Posizione** rispetto ai suoi operandi (o argomenti): prefisso, postfisso, infisso
- **Numero di operandi (arietà)**
  - Unari se hanno un solo operando
    - Esempi: `!a` `++a` `a--`
  - Binari se hanno due operandi
    - Esempi: `a && b` `a + b`
  - Ternari
    - Vedremo un esempio in seguito

Proprietà degli operatori 2/2

- **Precedenza** (o **priorità**) nell'ordine di esecuzione
  - Es: `1 + 2 * 3` è valutato come `1 + (2 * 3)`
  - `k<b+3` è valutato come `k<(b+3)`, e non `(k<b) +3`
- **Associatività**: ordine con cui vengono valutati due operatori con la stessa precedenza.
  - Associativi a sinistra: valutati da sinistra a destra
    - Es: `/` è associativo a sinistra, quindi `6/3/2` è uguale a `(6/3)/2`
  - Associativi a destra: valutati da destra a sinistra
    - Es: `=` è associativo a destra ...

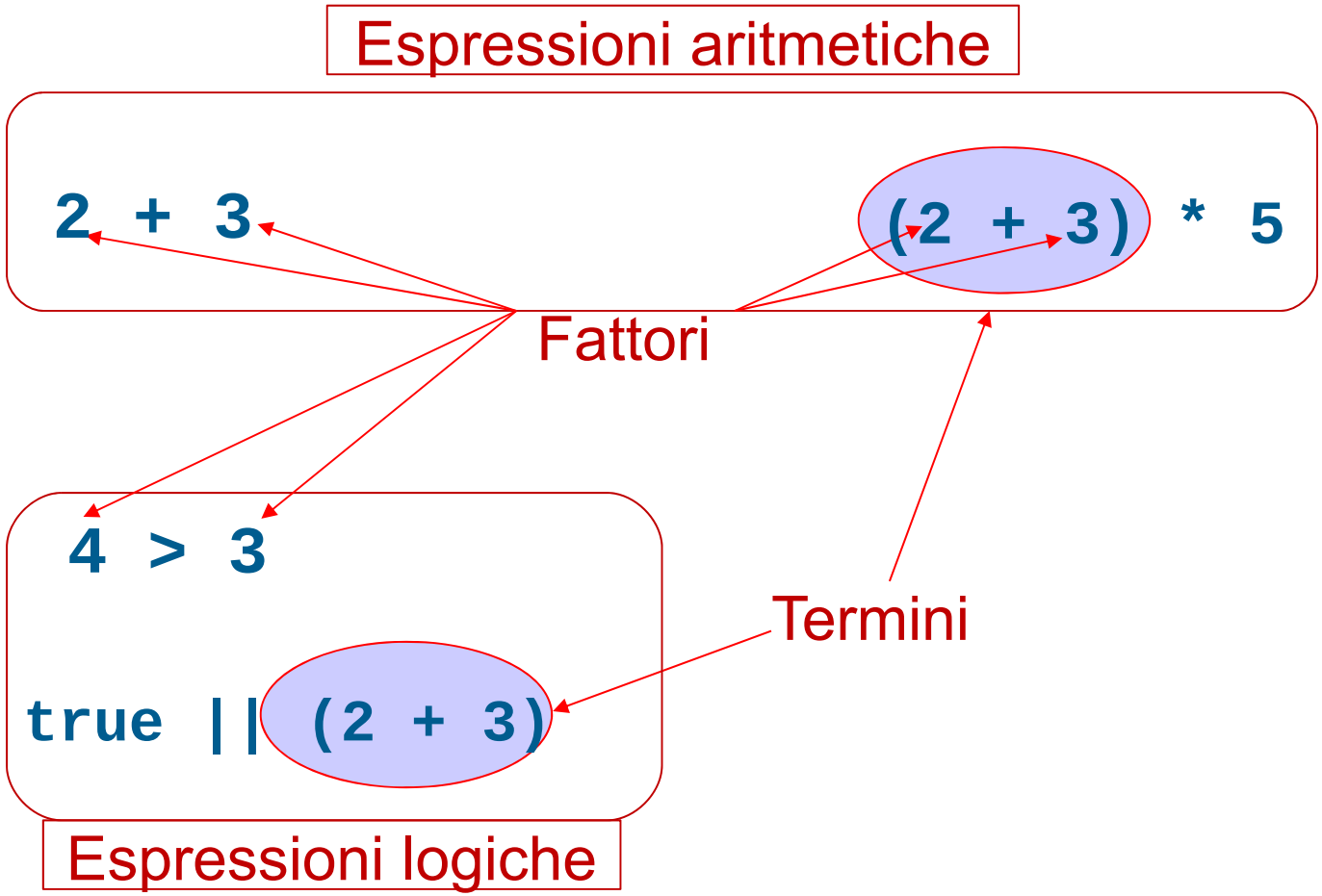
Associatività assegnamento

- L'operatore di assegnamento può comparire più volte in un'istruzione.
- L'associatività dell'operatore di assegnamento è a **destra**
- Esempio:
- `k = j = 5;`
- equivale a `k = (j = 5);` ossia:
- `j = 5;`
- `k = j;`
- Invece:
- `k = j + 2 = 5; // ERRORE !!!!!`
- perché `j + 2` non può fornire un **lvalue**, ossia l'indirizzo di una variabile!

Ordine valutazione espressioni

- Si calcolano prima i fattori, quindi i termini
  - **Fattori**: ottenuti dalle espressioni letterali e dal calcolo delle funzioni e degli operatori unari
  - **Termini**: ottenuti dal calcolo degli operatori binari
    - Moltiplicativi: * / %
    - Additivi: + -
    - Traslazione: << >>
    - Relazione: < > <= >=
    - Eguaglianza: == !=
    - Logici: && ||
    - Assegnamento: = += -= *= /=
- Con le parentesi possiamo modificare l'ordine di valutazione dei termini

Esempi


Espressioni aritmetiche
2 + 3
(2 + 3) * 5
Fattori
4 > 3
true || (2 + 3)
Termini
Espressioni logiche

Sintesi priorità degli operatori