

Sperimentazioni di Fisica I

mod. A – Lezione 9

Espressioni in C++ (CAP. 4)

*Dipartimento di Fisica e Astronomia “G. Galilei”,
Università degli Studi di Padova*

Espressioni in C++

- Un'espressione in C++ è composta da uno o più operandi e fornisce un risultato una volta valutata
- Gli operatori si dividono in categorie:
 - operatori unari, come l'*address-of* (&) e dereference (*) agiscono su un unico operando
 - operatori binari, come l'uguaglianza (==) e la moltiplicazione (*) che agiscono su due operandi
 - esiste anche un operatore ternario che richiede tre operandi
- per comprendere come le espressioni vengono valutate, è necessario capire la precedenza e associatività degli operatori

5 + 10 * 20 / 2

per poter determinare il risultato dell'espressione è necessario studiare come si comportano gli operatori che la compongono

Ordine di Valutazione

- ✓ la **precedenza** definisce come sono raggruppati gli operandi di un'espressione complessa

- ✗ nulla è specificato riguardo all'ordine di valutazione

```
int i = sqrt(val) * f2();
```

- ✓ le due **funzioni** saranno **invocate** prima della **moltiplicazione**, ma **non c'è** nessun modo di conoscere l'ordine di chiamata

```
int i = 0;  
cout << i << " " << ++i << endl;
```

- ✓ **non c'è** nessun modo per conoscere come verrà valutata l'istruzione :
 - A Il compilatore può valutare **prima ++i** e **poi i** producendo come **output 1 1**
 - B Il compilatore può valutare **prima i** e in tal caso avremo come **output 0 1**
 - C Il compilatore può **decidere un'altra strategia a noi sconosciuta**

Operatori Aritmetici

precedenza ↑	Associativity	Operator	Function	Use
	Left	+	unary plus	+ expr
	Left	-	unary minus	- expr
	Left	*	multiplication	expr * expr
	Left	/	division	expr / expr
	Left	%	remainder	expr % expr
	Left	+	addition	expr + expr
	Left	-	subtraction	expr - expr

- hanno tutti precedenza **LEFT → RIGHT**:

$$\begin{aligned} & 5 + 10 * 20 / 2 \\ = & 5 + (10 * 20) / 2 \\ & \quad \downarrow \\ = & 5 + 200 / 2 \\ = & 5 + (200 / 2) \\ & \quad \downarrow \\ = & 5 + 100 \\ = & 105 \end{aligned}$$

Divisione (/) e Modulo (%)

- Il risultato della divisione dipende dal tipo degli operandi

`int / int`

- `9 / 5`
- esegue `int` division $\Rightarrow 1$

`long / long`

- `9L / 5L`
- esegue `long` division $\Rightarrow 1$

`double / double`

- `9.0 / 5.0`
- esegue `double` division $\Rightarrow 1.8$

`float / float`

- `9.0F / 5.0F`
- esegue `float` division $\Rightarrow 1.8$

- Il modulo (%) ritorna il resto della divisione
- Può essere usato solo per tipi interi

`21 % 6 // risultato = 3`

`21 % 7 // risultato = 0`

`-21 % -8 // risultato = -5`

`21 % -5 // risultato = 1`

❖ Regola generale: `m % (-n) = m % n` e `(-m) % n = -(m % n)`

Operatori Logici e Relazionali

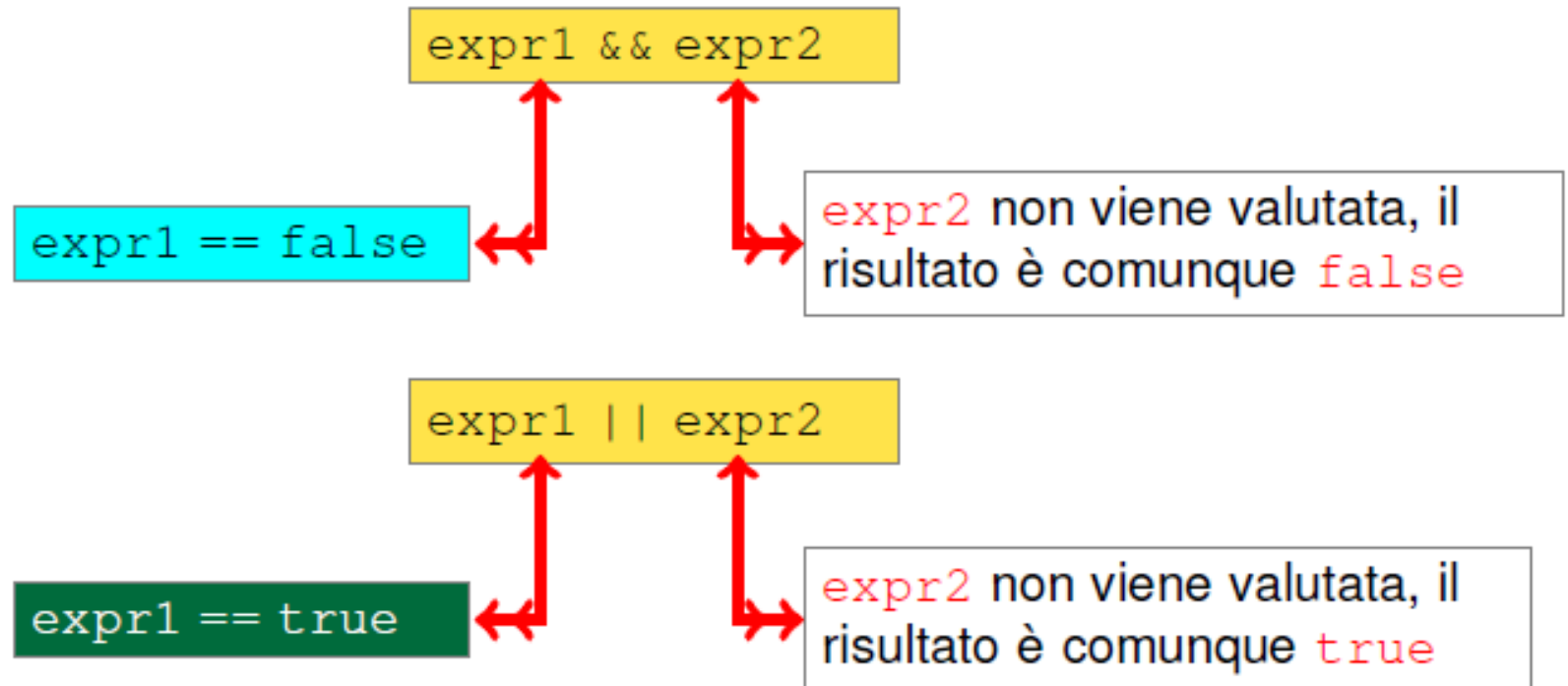
precedenza ↑	Associativity	Operator	Function	Use
	Right	!	logical NOT	!expr
	Left	<	less than	expr < expr
	Left	<=	less than or equal	expr <= expr
	Left	>	greater than	expr > expr
	Left	>=	greater than or equal	expr >= expr
	Left	==	equality	expr == expr
	Left	!=	inequality	expr != expr
	Left	&&	logical AND	expr && expr
	Left		logical OR	expr expr

- ✓ Gli operatori relazionali <, <=, >, >= si applicano a tutti i tipi aritmetici e ai puntatori
- ✓ Gli operatori logici &&, ||, ! si applicano a tutti i tipi convertibili in bool

Gli Operatori Logici: AND e OR

- ✓ AND è `true` se entrambi gli operandi sono `true`, altrimenti è `false`
- ✓ OR è `true` se almeno uno dei due è `true`; `false` se entrambi `false`

❖ Valutazione SHORT-CIRCUIT :



L'Operatore Logico NOT (!)

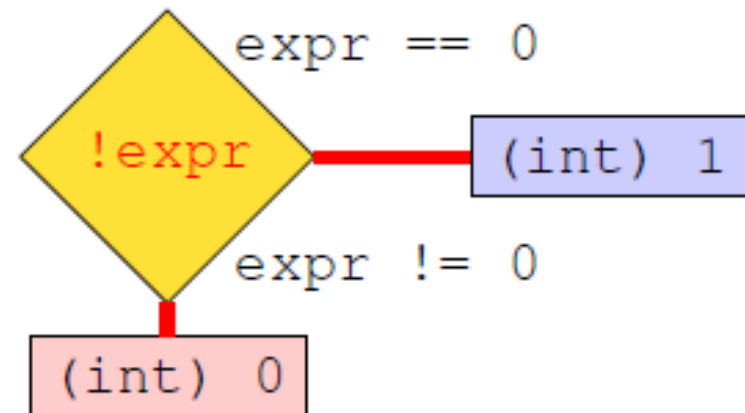
- L'operatore di negazione, **unario** è indicato con il simbolo **!**

Esempi:

`!a`

`!(x + 7.7)`

`!(a < b || c + d)`



- ✓ L'operatore **!** è diverso dall'operatore **NOT** dell'algebra booleana:
- ✓ soltanto **nell'algebra booleana**

$$\text{NOT} (\text{NOT } a) = a$$

➡ in C/C++

$$!!5 = 1$$



$$!(!5) = !(0) = 1$$

Disuguaglianze in C++

- In matematica si può scrivere $3 < j < 5$
per indicare che la variabile j assume valori compresi tra 3 e 5.

✓ Se $j=4 \Rightarrow 3 < j < 5$ è `true`

✓ Se $j=7 \Rightarrow 3 < j < 5$ è `false`

- In C++

$3 < j < 5$ è equivalente a $(3 < j) < 5$
quindi, per $j = 7$

$$3 < j < 5 \Rightarrow (3 < j) < 5$$

↓

$$1 < 5 \Rightarrow 1 \Rightarrow \text{true}$$

- La maniera corretta per implementare il test è

$$(3 < j) \ \&\& \ (j < 5)$$

L'espressione è `true` solo se entrambe le parentesi sono `true`

$$\begin{array}{ccccc} (3 < j) & \&\& & (j < 5) \\ \text{true} & \&\& & \text{false} \Rightarrow \text{false} \end{array}$$

Operatore di Assegnazione (=)

- ✓ È l'operatore con la **precedenza più bassa** di tutti
- ✓ è **RIGHT associative** :

la scrittura: `x = y = z = 0;`
equivale a: `(x = (y = (z = 0)))`;

- ✓ L'espressione

`x == y`  Test di eguaglianza

- ✓ è molto simile a

`x = y`  Statement di assegnazione

- ma il loro effetto è **radicalmente diverso!**

Errore comune nella scrittura di codice

```
if ( a = 1)
...
invece di
if ( a == 1)
...
```

- ❖ A volte molto difficile da individuare!!

Incremento (++) e Decremento (--)

✓ forniscono una **notazione convenzionale abbreviata** per **aggiungere (++)** e **sottrarre (-)** una unità da un oggetto di tipo intero

✓ possono essere utilizzati in **due forme** :

• **prefisso** : **++x, --x**

• **postfisso** : **x++, x--**

```
int x = 5;
int y = ++x;
cout << "x = " << x <<
      "y = " << y << endl;
```

- incrementa e poi assegna il valore

x = 6 e y = 6

```
int z = 5;
int y = z++;
cout << "z = " << z <<
      "y = " << y << endl;
```

- Assegna il valore e quindi incrementa

z = 6 e y = 5

Usate operatori POSTFISSI soltanto quando strettamente necessario

L'Operatore Condizionale (? :)

✓ È un operatore ternario

• Sintassi:

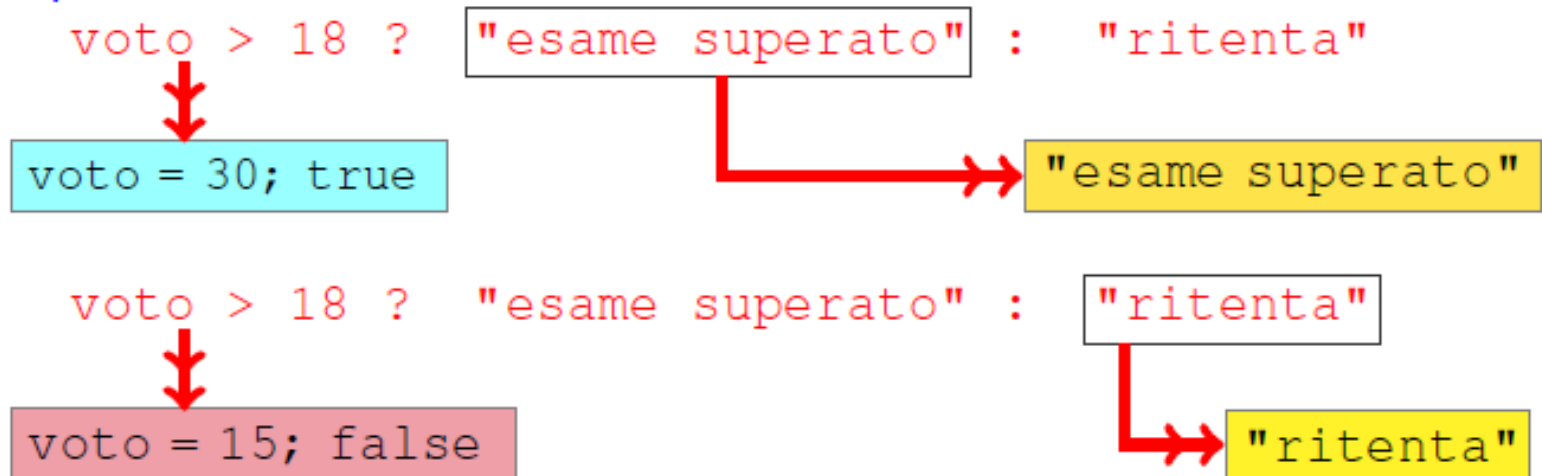
`expression1 ? expression2 : expression 3`

✓ Funzionamento:

Se `expression1` è `true`, allora il valore dell'espressione condizionale è quello di `expression2`.

Altrimenti, il valore dell'espressione è quello di `expression3`

• Esempi :



Conversioni tra Tipi

- ✓ Tutti gli operazioni vengono effettuate tra operandi dello stesso tipo
- ✓ se i tipi sono diversi, viene attuata una conversione

```
int i = 3.541 + 3;
```

- i due operandi della somma sono diversi:
`double + int` \Rightarrow `int` viene promosso a `double`
- la somma viene effettuata in virgola mobile ed il risultato viene trasformato in `int` ed usato per inizializzare la variabile `i`
- le conversioni prendono il nome di *conversioni implicite*
- Quando avvengono ?
 - tipi interi più piccoli di `int` sono convertiti all'intero più grande
 - in espressioni relazionali, `non-bool` sono convertiti a `bool`
 - in inizializzazioni, l'inizializzatore è convertito al tipo della variabile
 - in espressioni aritmetiche e relazionali con tipi misti, i tipi sono convertiti ad un tipo comune

Conversioni in Espressioni

Promozioni tra interi

1. `char`, `short` (signed and unsigned) sono promossi a `int`
2. se dopo la prima conversione l'**espressione** è di **tipo misto**: l'operando di tipo inferiore è promosso al tipo superiore secondo le regole:
3. nel caso di operandi `signed` e `unsigned`, il tipo con meno bit viene convertito a quello con più bit
4. se la **dimensione** è la stessa, il tipo `signed` è convertito a quello `unsigned`

Regole di conversione:

1. `long long double l;`
2. `long double`
3. `double d;`
4. `float f;`
5. `unsigned long`
6. `unsigned u;`
7. `long l;`
8. `int i;`
9. `char c, short s;`

promozione

Esempi di Conversioni

```
bool  Bflag;    char Cval;  
short Sval;     unsigned short USval;  
int   Ival;     unsigned int   Uival;  
long  Lval;     unsigned long  ULval;  
float Fval;     double          Dval;
```

```
3.1419 + 'a'; // 'a' -> int, then int -> double
```

```
Dval + Ival; // Ival, int -> double
```

```
Dval + Fval; // Fval, float -> double
```

```
Ival = Dval; // Dval, double -> int (truncation)
```

```
Bflag = Dval; // if Dval is 0 -> false, else true
```

```
Cval + Fval; // Cval, char -> int, then int -> float
```

```
Sval + Cval; // Sval, short -> int, Cval, char -> int
```

```
Cval + Lval; // Cval, char -> long
```

```
Ival + ULval; // Ival, int -> unsigned long
```

```
USval + Ival; // dipende dalle dimensioni di unsigned short e int
```

```
Uival + Lval; // dipende dalle dimensioni di unsigned int e long
```