Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Espressioni ed operatori

Catia Prandi - catia.prandi2@unibo.it

Credit: Pietro Di Lena

Q. What's the difference between C and C++?
A. Nobody knows, since C - C++ is undefined.

Espressioni e operatori

- Gli **operatori** e gli **operandi** sono *elementi sintattici* del linguaggio C.
- Una **espressione** è una combinazione di operandi e operatori:

```c
int x;
// Operatore: =, Operandi: x e 0
x = 0;

// Operatore: +, Operandi: x e 1
// Operatore: =, Operandi x e x+1
x = x + 1;
```

Espressioni e operatori

- Gli **operatori** e gli **operandi** sono *elementi sintattici* del linguaggio C.
- Una **espressione** è una combinazione di operandi e operatori:

```c
int x;
// Operatore: =, Operandi: x e 0
x = 0;

// Operatore: +, Operandi: x e 1
// Operatore: =, Operandi x e x+1
x = x + 1;
```

- Distinguiamo due tipi differenti di **valutazioni** che possono essere effettuate dal compilatore per ogni espressione/sotto-espressione
  - **calcolo del valore**: calcolo del valore dell'espressione.
    - Ex. l'espressione `x+1;`, calcola un valore;
  - **side-effect**: modifica del valore di un oggetto nell'espressione.
    - Ex. l'espressione `x=0;`, modifica il contenuto della variabile `x`).

Espressioni e operatori

- Gli **operatori** e gli **operandi** sono *elementi sintattici* del linguaggio C.
- Una **espressione** è una combinazione di operandi e operatori:

```
int x;
// Operatore: =, Operandi: x e 0
x = 0;

// Operatore: +, Operandi: x e 1
// Operatore: =, Operandi x e x+1
x = x + 1;
```

- Distinguiamo due tipi differenti di **valutazioni** che possono essere effettuate dal compilatore per ogni espressione/sotto-espressione
  - **calcolo del valore**: calcolo del valore dell'espressione.
    - Ex. l'espressione `x+1;`, calcola un valore;
  - **side-effect**: modifica del valore di un oggetto nell'espressione.
    - Ex. l'espressione `x=0;`, modifica il contenuto della variabile `x`).
- Ad una espressione (anche quelle di assegnamento) è associato un **tipo** e di conseguenza un **valore**.
  - Ex. Il valore associato all'espressione `x=0;` è il valore assegnato ad `x`.
  - Ex. Il valore associato all'espressione `x+1;` è il risultato dell'operazione di somma.
- E' possibile definire espressioni con tipo `void`, che non hanno quindi un valore associato.

Operatori

- Il linguaggio C mette a disposizione un vasto numero di operatori:
  - operatori aritmetici
  - operatori relazionali
  - operatori logici
  - operatori di assegnamento
  - operatori bitwise (bit a bit)
  - operatori di accesso e puntatori (che non vedremo adesso).
  - altri operatori, tra cui:
    - operatore condizionale,
    - operatore di concatenazione,
    - operatorei di conversione di tipo,
    - operatore di indirizzo.
- Gli operatori C possono essere **unari** (un solo operando), **binari** (due operandi), **ternari** (tre operandi).
- Gli operatori soggetti a regole di **associatività**, **precedenza** e **ordine di valutazione**.

Precedenza e associatività

- Tutti gli operatori sono soggetti a ben precise regole di **precedenza** e **associatività**.
- Ad esempio, senza precise regole di precedenza e associatività, l'espressione

```
1 - 2 * 2 - 1
```

potrebbe essere interpretata in modi differenti:

Precedenza e associatività

- Tutti gli operatori sono soggetti a ben precise regole di **precedenza** e **associatività**.
- Ad esempio, senza precise regole di precedenza e associatività, l'espressione

```
1 - 2 * 2 - 1
```

potrebbe essere interpretata in modi differenti:

- Se assumiamo che l'operatore - abbia priorità maggiore rispetto all'operatore *:

```
(1 - 2) * (2 - 1) ⇒ −1
```

Precedenza e associatività

- Tutti gli operatori sono soggetti a ben precise regole di **precedenza** e **associatività**.
- Ad esempio, senza precise regole di precedenza e associatività, l'espressione

  `1 - 2 * 2 - 1`

  potrebbe essere interpretata in modi differenti:
  - Se assumiamo che l'operatore - abbia priorità maggiore rispetto all'operatore *:

    `(1 - 2) * (2 - 1)` $\Rightarrow -1$
  - Se assumiamo che l'operatore - sia associativo da destra a sinistra:

    `1 - ((2 * 2) - 1)` $\Rightarrow -2$

Precedenza e associatività

- Tutti gli operatori sono soggetti a ben precise regole di **precedenza** e **associatività**.
- Ad esempio, senza precise regole di precedenza e associatività, l'espressione

  `1 - 2 * 2 - 1`

  potrebbe essere interpretata in modi differenti:
  - Se assumiamo che l'operatore - abbia priorità maggiore rispetto all'operatore *:

    `(1 - 2) * (2 - 1)` $\Rightarrow -1$
  - Se assumiamo che l'operatore - sia associativo da destra a sinistra:

    `1 - ((2 * 2) - 1)` $\Rightarrow -2$
  - Interpretazione standard:
    - l'operatore * ha priorità maggiore dell'operatore -
    - l'operatore - è associativo da sinistra verso destra

    `(1 - (2 * 2)) - 1` $\Rightarrow -4$

Precedenza e associatività degli operatori in C

Operatore	Descrizione	Esempio	Associatività	Precedenza
++ -	Post incremento/decremento	x++ x-	⟹	1
()	Chiamata a funzione	x(y)		
[ ]	Elemento di un array	x[y]		
.	Membro di struttura o union	x.y		
->	Puntatore a membro di struttura	x->y		
!	NOT logico	!x	⟸	2
~	Complemento ad uno	~x		
++ -	Pre incremento/decremento	++x -x		
+ -	Operatori unari di segno	+x -x		
&	Operatore di indirizzo	&x		
*	Operatore di deferenziazione	*x		
(type)	Operatore di conversione di tipo (cast)	(type)x		
sizeof	Dimensione in byte	sizeof(x)		
*	Moltiplicazione	x * y	⟹	3
/	Divisione	x / y		
%	Resto della divisione intera	x % y		
+	Somma	x + y	⟹	4
-	Sottrazione	x - y		
«	Shift verso sinistra	x « y	⟹	5
»	Shift verso destra	x » y		
<	Minore	x < y	⟹	6
<=	Minore o uguale	x <= y		
>	Maggiore	x > y		
>=	Maggiore o uguale	x >= y		
==	Uguaglianza	x == y	⟹	7
!=	Non uguaglianza	x != y		
&	AND bit-a-bit	x & y	⟹	8
^	XOR bit-a-bit	x ^ y	⟹	9
|	OR bit-a-bit	x | y	⟹	10
&&	AND logico	x && y	⟹	11
||	OR logico	x || y	⟹	12
? :	Operatore condizionale	x ? y : z	⟸	13
=	Assegnamento	x = y	⟸	14
*= /= %= += -= ^= «= »= &= |=	Assegnamenti composti	x*=y x%=y x+=y x-=y x^=y x«=y x»=y x&=y x|=y		
,	Operatore di concatenazione (virgola)	x, y	⟹	15

Ordine di valutazione

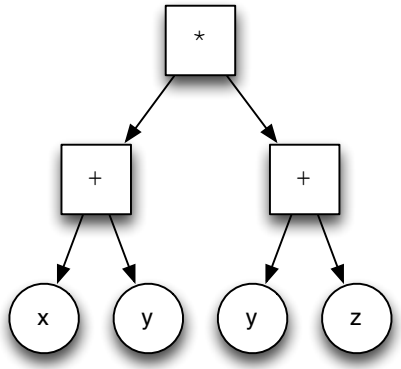
- In C, come per molti altri linguaggi di programmazione, l'**ordine di valutazione** degli operatori è generalmente **non specificato**.
  - Le eccezioni verranno esplicitamente indicate.

Ordine di valutazione

- In C, come per molti altri linguaggi di programmazione, l'**ordine di valutazione** degli operatori è generalmente **non specificato**.
  - Le eccezioni verranno esplicitamente indicate.
- Ad esempio, nell'espressione

```
(x+y)*(y+z)
```

$\Longrightarrow$



per l'operatore * non possiamo assumere che la sotto espressione x+y venga valutata prima di y+z. E per l'operatore + non possiamo assumere che x venga valutato prima di y (risp. y prima di z).

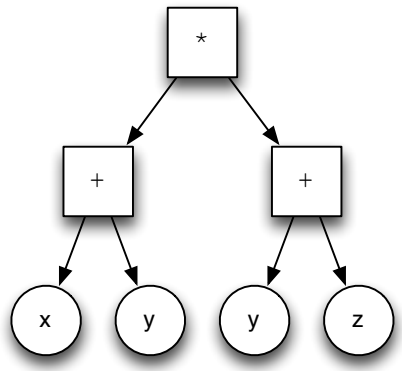
- Motivazione: lasciare ai compilatori la libertà di ottimizzare opportunamente le istruzioni coinvolte nella valutazione di espressioni.

Ordine di valutazione

- In C, come per molti altri linguaggi di programmazione, l'**ordine di valutazione** degli operatori è generalmente **non specificato**.
  - Le eccezioni verranno esplicitamente indicate.
- Ad esempio, nell'espressione

```
(x+y)*(y+z)
```

$\Longrightarrow$



per l'operatore * non possiamo assumere che la sotto espressione x+y venga valutata prima di y+z. E per l'operatore + non possiamo assumere che x venga valutato prima di y (risp. y prima di z).

- Motivazione: lasciare ai compilatori la libertà di ottimizzare opportunamente le istruzioni coinvolte nella valutazione di espressioni.
- Formalmente, lo standard ISO definisce i **sequence point**: *"punto del programma in cui è garantito che tutti i* **side effects** *delle istruzioni precedenti siano stati eseguiti"*.
  - Ex. Abbiamo un sequence point sul punto e virgola che termina un'espressione.
  - Prima di passare a valutare l'espressione successiva, siamo sicuri che tutti i side effects che precedono il punto e virgola sono stati eseguiti.

Operatori aritmetici

Operatore	Descrizione	Tipo	Esempio
*	Moltiplica il primo e il secondo operando	Binario	x * y
/	Divide il primo operando per il secondo		x / y
%	Calcola il resto della divisione intera tra il primo e secondo operando		x % y
+	Calcola la somma tra i due operandi		x + y
-	Calcola la sottrazione tra i due operandi		x - y
-	Inverte il segno dell'operando	Unario	-x
+	Non ha effetto, in quanto non modifica il segno dell'operando		+x
++	Incrementa di una unità l'operando dopo avere restituito il valore		x++
-	Decrementa di una unità l'operando dopo avere restituito il valore		x-
++	Incrementa di una unità l'operando prima di averne restituito il valore		++x
-	Decrementa di una unità l'operando prima di averne restituito il valore		-x

- Tutti gli operatori aritmetici, tranne %, possono essere usati sia con i tipi di dato interi che con i tipi in virgola mobile.
- L'operatore % può essere utilizzato solo con tipi di dato intero (in caso contrario, il programma non viene compilato).
- Se il secondo operando è 0, le operazioni aritmetiche di divisione (/) e resto della divisione intera (%) sono non definite. In tali casi, il programma in esecuzione potrebbe essere terminato.
- L'operatore / (divisione) viene inteso come *divisione intera* quando entrambi gli operandi sono interi (il risultato viene troncato ad un intero).

Operatori aritmetici: incremento e decremento

- In C esistono due operatori unari *speciali* (++ e --) per **incrementare** e **decrementare** di un'unità la variabile a cui sono applicati.

Operatori aritmetici: incremento e decremento

- In C esistono due operatori unari *speciali* (++ e --) per **incrementare** e **decrementare** di un'unità la variabile a cui sono applicati.
- Possono essere applicati in due modalità:
  - **prefissa**: la variabile viene modificata **prima** di utilizzarne il valore

```
1 int x = 0, y = 0, z = 0;
2 ++x;       // x = x+1=1
3 --y;       // y = y-1 =-1
4 z = ++x; // z = x = 2
```

  - **postfissa**: la variabile viene modificata **dopo** averne utilizzato il valore

```
1 int x=0, y=0, z=0;
2 x++;       // x=x+1=1
3 y--;       // y=y-1=-1
4 z = x++; // z=1, x=2
```

Operatori aritmetici: incremento e decremento

- In C esistono due operatori unari *speciali* (++ e --) per **incrementare** e **decrementare** di un'unità la variabile a cui sono applicati.
- Possono essere applicati in due modalità:
  - **prefissa**: la variabile viene modificata **prima** di utilizzarne il valore

```
1 int x = 0, y = 0, z = 0;
2 ++x;       // x = x+1=1
3 --y;       // y = y-1 =-1
4 z = ++x; // z = x = 2
```

  - **postfissa**: la variabile viene modificata **dopo** averne utilizzato il valore

```
1 int x=0, y=0, z=0;
2 x++;       // x=x+1=1
3 y--;       // y=y-1=-1
4 z = x++; // z=1, x=2
```

- Il vantaggio di impiegare gli operatori di incremento e decremento risiede nel fatto di rendere il codice più compatto. Ad esempio,

```
y = ++x;
```
$\Longrightarrow$
```
x = x+1;
y = x;
```

```
y = x++;
```
$\Longrightarrow$
```
y = x;
x = x+1;
```

Incremento/decremento e **side effects**

- Gli operatori di incremento e decremento provocano dei **side effect** nelle espressioni: oltre a produrre un valore, **modificano lo stato di una variabile**.
- In alcuni casi, i side effects determinano espressioni la cui semantica non è definita.
- Esempio: qual è il valore della variabili `x` e `y` dopo l'istruzione a riga 2?

```
1 int x = 1;
2 int y = ++x * x;
```

Incremento/decremento e **side effects**

- Gli operatori di incremento e decremento provocano dei **side effect** nelle espressioni: oltre a produrre un valore, **modificano lo stato di una variabile**.
- In alcuni casi, i side effects determinano espressioni la cui semantica non è definita.
- Esempio: qual è il valore della variabili `x` e `y` dopo l'istruzione a riga 2?

```
1 int x = 1;
2 int y = ++x * x;
```

Risposta: non definito. Ricordiamo che l'ordine di valutazione degli operatori in C è **non specificato** (se non esplicitamente indicato):

- se `++x` è valutato prima di `x`, allora *potremmo avere* `y=2*2=4, x=2`;
- se `x` è valutato prima di `++x`, allora *potremmo avere* `y=2*1=2, x=2`.

Incremento/decremento e **side effects**

- Gli operatori di incremento e decremento provocano dei **side effect** nelle espressioni: oltre a produrre un valore, **modificano lo stato di una variabile**.
- In alcuni casi, i side effects determinano espressioni la cui semantica non è definita.
- Esempio: qual è il valore della variabili `x` e `y` dopo l'istruzione a riga 2?

```
1 int x = 1;
2 int y = ++x * x;
```

Risposta: non definito. Ricordiamo che l'ordine di valutazione degli operatori in C è **non specificato** (se non esplicitamente indicato):

- se `++x` è valutato prima di `x`, allora *potremmo avere* `y=2*2=4, x=2`;
- se `x` è valutato prima di `++x`, allora *potremmo avere* `y=2*1=2, x=2`.

- Esempio: qual è il valore della variabile `x` dopo l'istruzione a riga 2?

```
1 int x=1;
2 x = x++;
```

Incremento/decremento e **side effects**

- Gli operatori di incremento e decremento provocano dei **side effect** nelle espressioni: oltre a produrre un valore, **modificano lo stato di una variabile**.
- In alcuni casi, i side effects determinano espressioni la cui semantica non è definita.
- Esempio: qual è il valore della variabili `x` e `y` dopo l'istruzione a riga 2?

```
1 int x = 1;
2 int y = ++x * x;
```

Risposta: non definito. Ricordiamo che l'ordine di valutazione degli operatori in C è **non specificato** (se non esplicitamente indicato):

- se `++x` è valutato prima di `x`, allora *potremmo avere* `y=2*2=4, x=2;`
- se `x` è valutato prima di `++x`, allora *potremmo avere* `y=2*1=2, x=2.`

- Esempio: qual è il valore della variabile `x` dopo l'istruzione a riga 2?

```
1 int x=1;
2 x = x++;
```

Risposta: non definito. Potrebbe essere *interpretata* dal compilatore come

```
1 int x = 1, tmp;
2 tmp = x; // Valuta x
3 x = x+1; // Incrementa x
4 x = tmp; // Assegna ad x
5 // Qui x = 1
```

oppure

```
1 int x = 1, tmp;
2 tmp = x; // Valuta x
3 x = tmp; // Assegna ad x
4 x = x+1; // Incrementa x
5 // Qui x = 2
```

Parentesi: comportamento *non definito*, *non specificato* e *definito dall'implementazione*

- Alcuni aspetti della semantica del linguaggio C non sono esplicitamente trattati dallo standard. Abbiamo la seguente terminologia:
  - **Implementation-defined behavior** (comportamento definito dall'implementazione). L'implementazione deve scegliere un comportamento per un particolare costrutto sintattico e questo deve essere documentato. La compilazione deve andare a buon fine.

Parentesi: comportamento *non definito*, *non specificato* e *definito dall'implementazione*

- Alcuni aspetti della semantica del linguaggio C non sono esplicitamente trattati dallo standard. Abbiamo la seguente terminologia:
  - **Implementation-defined behavior** (comportamento definito dall'implementazione). L'implementazione deve scegliere un comportamento per un particolare costrutto sintattico e questo deve essere documentato. La compilazione deve andare a buon fine.
  - **Unspecified behavior** (comportamento non specificato). Come *implementation-defined behavior* ma l'implementazione non è tenuta a documentare la scelta.

Parentesi: comportamento *non definito*, *non specificato* e *definito dall'implementazione*

- Alcuni aspetti della semantica del linguaggio C non sono esplicitamente trattati dallo standard. Abbiamo la seguente terminologia:
  - **Implementation-defined behavior** (comportamento definito dall'implementazione). L'implementazione deve scegliere un comportamento per un particolare costrutto sintattico e questo deve essere documentato. La compilazione deve andare a buon fine.
  - **Unspecified behavior** (comportamento non specificato). Come *implementation-defined behavior* ma l'implementazione non è tenuta a documentare la scelta.
  - **Undefined behavior** (comportamento non definito). Lo standard non impone alcun requisito particolare sul costrutto sintattico. L'implementazione è libera di scegliere qualsiasi comportamento. Ad esempio, lo standard non impone nessun tipo di restrizione all'implementazione (di compilatori) sul valore da assegnare a variabili non inizializzate. Il comportamento di programmi che dipendono da istruzioni con semantica non definita è **impredicibile**.

Parentesi: comportamento *non definito*, *non specificato* e *definito dall'implementazione*

- Alcuni aspetti della semantica del linguaggio C non sono esplicitamente trattati dallo standard. Abbiamo la seguente terminologia:
  - **Implementation-defined behavior** (comportamento definito dall'implementazione). L'implementazione deve scegliere un comportamento per un particolare costrutto sintattico e questo deve essere documentato. La compilazione deve andare a buon fine.
  - **Unspecified behavior** (comportamento non specificato). Come *implementation-defined behavior* ma l'implementazione non è tenuta a documentare la scelta.
  - **Undefined behavior** (comportamento non definito). Lo standard non impone alcun requisito particolare sul costrutto sintattico. L'implementazione è libera di scegliere qualsiasi comportamento. Ad esempio, lo standard non impone nessun tipo di restrizione all'implementazione (di compilatori) sul valore da assegnare a variabili non inizializzate. Il comportamento di programmi che dipendono da istruzioni con semantica non definita è **impredicibile**.
- E' buona norma evitare di scrivere del codice che possa produrre comportamenti *non definiti* o *non specificati*: il comportamento del programma potrebbe essere differente a seconda del compilatore utilizzato per generare l'eseguibile.
- E' buona norma anche evitare di scrivere codice dipendente da una specifica implementazione, se l'obiettivo è sviluppare programmi portabili.
- Alcuni compilatori segnalano la presenza di codice la cui semantica è *non definita*.

Espressioni con semantica non definita: come evitare i problemi

- Le operazioni di post-incremento/decremento eseguono l'operazione di incremento **dopo** aver utilizzato il valore della variabile.
- Quello che lo standard ISO C89 non definisce è **quando** deve essere effettuata l'operazione di incremento/decremento: può avvenire in un qualsiasi momento compreso tra la *valutazione del valore della variabile* e la *fine della valutazione dell'espressione* (più precisamente, il *sequence point* successivo).

Espressioni con semantica non definita: come evitare i problemi

- Le operazioni di post-incremento/decremento eseguono l'operazione di incremento **dopo** aver utilizzato il valore della variabile.
- Quello che lo standard ISO C89 non definisce è **quando** deve essere effettuata l'operazione di incremento/decremento: può avvenire in un qualsiasi momento compreso tra la *valutazione del valore della variabile* e la *fine della valutazione dell'espressione* (più precisamente, il *sequence point* successivo).
- Le regole per comprendere quando ci troviamo a definire espressioni non definite sono alquanto criptiche. Citiamo direttamente lo standard ISO C89:
  Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.

Espressioni con semantica non definita: come evitare i problemi

- Le operazioni di post-incremento/decremento eseguono l'operazione di incremento **dopo** aver utilizzato il valore della variabile.
- Quello che lo standard ISO C89 non definisce è **quando** deve essere effettuata l'operazione di incremento/decremento: può avvenire in un qualsiasi momento compreso tra la *valutazione del valore della variabile* e la *fine della valutazione dell'espressione* (più precisamente, il *sequence point* successivo).
- Le regole per comprendere quando ci troviamo a definire espressioni non definite sono alquanto criptiche. Citiamo direttamente lo standard ISO C89:
  Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.
- Come leggiamo le indicazioni dello standard:
  1. Il valore di una variabile deve essere modificato al più una volta nella stessa espressione (o meglio, *tra due sequence point successivi*).

Espressioni con semantica non definita: come evitare i problemi

- Le operazioni di post-incremento/decremento eseguono l'operazione di incremento **dopo** aver utilizzato il valore della variabile.
- Quello che lo standard ISO C89 non definisce è **quando** deve essere effettuata l'operazione di incremento/decremento: può avvenire in un qualsiasi momento compreso tra la *valutazione del valore della variabile* e la *fine della valutazione dell'espressione* (più precisamente, il *sequence point* successivo).
- Le regole per comprendere quando ci troviamo a definire espressioni non definite sono alquanto criptiche. Citiamo direttamente lo standard ISO C89:
  Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.
- Come leggiamo le indicazioni dello standard:
  1. Il valore di una variabile deve essere modificato al più una volta nella stessa espressione (o meglio, *tra due sequence point successivi*).
  2. Se il valore di una variabile `x` viene modificato in una espressione (o meglio, *tra due sequence point successivi*), allora ulteriori riferimenti alla stessa variabile `x` nell'espressione devono servire solo per il calcolo del valore da assegnare ad `x`.

Espressioni con semantica non definita: come evitare i problemi

- Le operazioni di post-incremento/decremento eseguono l'operazione di incremento **dopo** aver utilizzato il valore della variabile.
- Quello che lo standard ISO C89 non definisce è **quando** deve essere effettuata l'operazione di incremento/decremento: può avvenire in un qualsiasi momento compreso tra la *valutazione del valore della variabile* e la *fine della valutazione dell'espressione* (più precisamente, il *sequence point* successivo).
- Le regole per comprendere quando ci troviamo a definire espressioni non definite sono alquanto criptiche. Citiamo direttamente lo standard ISO C89:
  Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.
- Come leggiamo le indicazioni dello standard:
  1. Il valore di una variabile deve essere modificato al più una volta nella stessa espressione (o meglio, *tra due sequence point successivi*).
  2. Se il valore di una variabile `x` viene modificato in una espressione (o meglio, *tra due sequence point successivi*), allora ulteriori riferimenti alla stessa variabile `x` nell'espressione devono servire solo per il calcolo del valore da assegnare ad `x`.
  3. Se ci restringiamo ai soli operatori di incremento la *regola* 2 diventa come segue. Se in una espressione (o meglio, *tra due sequence point successivi*) si utilizza l'operatore di incremento/decremento su una variabile, non utilizzare in altro modo la stessa variabile nell'espressione.

Operatori relazionali

Operatore	Descrizione	Tipo	Esempio
==	1 se i due operandi hanno lo stesso valore, 0 altrimenti	Binario	x == y
!=	1 se i due operandi hanno valore differente, 0 altrimenti		x != y
>	1 se il primo operando ha valore maggiore del secondo, 0 altrimenti		x > y
>=	1 se il primo operando ha valore maggiore o uguale al secondo, 0 altrimenti		x >= y
<	1 se il primo operando ha valore minore del secondo, 0 altrimenti		x < y
<=	1 se il primo operando ha valore minore o uguale al secondo, 0 altrimenti		x <= y

- Gli **operatori relazionali** permettono il confronto dei valori di due espressioni.
- Generano un risultato **logico**: vero (1) o falso (0) di tipo `int`.

Operatori relazionali

Operatore	Descrizione	Tipo	Esempio
==	1 se i due operandi hanno lo stesso valore, 0 altrimenti	Binario	x == y
!=	1 se i due operandi hanno valore differente, 0 altrimenti		x != y
>	1 se il primo operando ha valore maggiore del secondo, 0 altrimenti		x > y
>=	1 se il primo operando ha valore maggiore o uguale al secondo, 0 altrimenti		x >= y
<	1 se il primo operando ha valore minore del secondo, 0 altrimenti		x < y
<=	1 se il primo operando ha valore minore o uguale al secondo, 0 altrimenti		x <= y

- Gli **operatori relazionali** permettono il confronto dei valori di due espressioni.
- Generano un risultato **logico**: vero (1) o falso (0) di tipo `int`.

```
1 int x = 3, y = 0;
2 int z = x==y; // z=0
```

```
1 int x = 3, y = 0;
2 int z = y<=x; // z=1
```

```
1 int x = 3, y=0;
2 int z = x!=y; // z=1
```

Operatori relazionali

Operatore	Descrizione	Tipo	Esempio
==	1 se i due operandi hanno lo stesso valore, 0 altrimenti	Binario	x == y
!=	1 se i due operandi hanno valore differente, 0 altrimenti		x != y
>	1 se il primo operando ha valore maggiore del secondo, 0 altrimenti		x > y
>=	1 se il primo operando ha valore maggiore o uguale al secondo, 0 altrimenti		x >= y
<	1 se il primo operando ha valore minore del secondo, 0 altrimenti		x < y
<=	1 se il primo operando ha valore minore o uguale al secondo, 0 altrimenti		x <= y

- Gli **operatori relazionali** permettono il confronto dei valori di due espressioni.
- Generano un risultato **logico**: vero (1) o falso (0) di tipo `int`.

```
1 int x = 3, y = 0;
2 int z = x==y; // z=0
```

```
1 int x = 3, y = 0;
2 int z = y<=x; // z=1
```

```
1 int x = 3, y=0;
2 int z = x!=y; // z=1
```

- Attenzione: in C la relazione 3 > 2 > 1 non ha il significato matematico *standard* ma viene interpretata come:

Operatori relazionali

Operatore	Descrizione	Tipo	Esempio
==	1 se i due operandi hanno lo stesso valore, 0 altrimenti	Binario	x == y
!=	1 se i due operandi hanno valore differente, 0 altrimenti		x != y
>	1 se il primo operando ha valore maggiore del secondo, 0 altrimenti		x > y
>=	1 se il primo operando ha valore maggiore o uguale al secondo, 0 altrimenti		x >= y
<	1 se il primo operando ha valore minore del secondo, 0 altrimenti		x < y
<=	1 se il primo operando ha valore minore o uguale al secondo, 0 altrimenti		x <= y

- Gli **operatori relazionali** permettono il confronto dei valori di due espressioni.
- Generano un risultato **logico**: vero (1) o falso (0) di tipo `int`.

```c
int x = 3, y = 0;
int z = x==y; // z=0
```

```c
int x = 3, y = 0;
int z = y<=x; // z=1
```

```c
int x = 3, y=0;
int z = x!=y; // z=1
```

- Attenzione: in C la relazione $3 > 2 > 1$ non ha il significato matematico *standard* ma viene interpretata come:

$$3 > 2 > 1 \Rightarrow (3 > 2) > 1 \Rightarrow 1 > 1 \Rightarrow 0 \text{ (falso)}$$

applicando l'associatività da sinistra a destra.

- La stessa considerazione vale anche per gli altri operatori logici.

Conversione di tipo

- Abbiamo visto che sul calcolatore la codifica dei tipi di dato interi è alquanto differente dalla codifica dei tipo di dato in virgola mobile.
- Generalmente, i calcolatori hanno componenti hardware distinte per le operazioni tra dati in virgola mobile e dati di tipo intero.
- Come vengono gestite le espressioni che coinvolgono tipi di dati differenti?

Conversione di tipo

- Abbiamo visto che sul calcolatore la codifica dei tipi di dato interi è alquanto differente dalla codifica dei tipo di dato in virgola mobile.
- Generalmente, i calcolatori hanno componenti hardware distinte per le operazioni tra dati in virgola mobile e dati di tipo intero.
- Come vengono gestite le espressioni che coinvolgono tipi di dati differenti?
- La **conversione di tipo** è una operazione che permette di convertire una variabile da un tipo di dato ad un altro.
- In C abbiamo due differenti modalità di conversione tra tipi.
  - La **conversione implicita di tipo** (**coercion**) viene effettuata dal compilatore in presenza di espressioni che coinvolgono tipi di dato differente.
  - La **conversione esplicita di tipo** (**typecasting**) viene definita esplicitamente dal programmatore tramite un opportuno operatore (cast).

Conversione di tipo implicita: regole

- Durante la **valutazione** di un'espressione gli operandi riferiti ad uno stesso operatore vengono convertiti tutti nello stesso tipo prima di calcolare il valore dell'espressione.
- Detto in modo *approssimativo*, il tipo di *dimensione minore* viene convertito nel tipo di *dimensione maggiore*.

Conversione di tipo implicita: regole

- Durante la **valutazione** di un'espressione gli operandi riferiti ad uno stesso operatore vengono convertiti tutti nello stesso tipo prima di calcolare il valore dell'espressione.
- Detto in modo *approssimativo*, il tipo di *dimensione minore* viene convertito nel tipo di *dimensione maggiore*.
- Nello specifico, si utilizzano le regole note come **usual arithmetic conversions**:
  1. se un operando è `long double`, l'altro è convertito in `long double`;
  2. altrimenti, se un operando è `double`, l'altro è convertito in `double`;
  3. altrimenti, se un operando è `float`, l'altro è convertito in `float`;
  4. altrimenti le regole di **integral promotion** sono applicate su entrambi gli operandi.

Conversione di tipo implicita: regole

- Durante la **valutazione** di un'espressione gli operandi riferiti ad uno stesso operatore vengono convertiti tutti nello stesso tipo prima di calcolare il valore dell'espressione.
- Detto in modo *approssimativo*, il tipo di *dimensione minore* viene convertito nel tipo di *dimensione maggiore*.
- Nello specifico, si utilizzano le regole note come **usual arithmetic conversions**:
  1. se un operando è `long double`, l'altro è convertito in `long double`;
  2. altrimenti, se un operando è `double`, l'altro è convertito in `double`;
  3. altrimenti, se un operando è `float`, l'altro è convertito in `float`;
  4. altrimenti le regole di **integral promotion** sono applicate su entrambi gli operandi.
- Regole di **integral promotion** (*promozione ad intero*):
  A. se un operando è `long unsigned int`, l'altro è convertito in `long unsigned int`;
  B. altrimenti, se un operando è `long int` e l'altro `unsigned int`:
     1. se `long int` può rappresentare tutti i valori di `unsigned int`, l'operando `unsigned int` è convertito in `long int`;
     2. altrimenti, entrambi sono convertiti in `long unsigned int`.
  C. altrimenti, se un operando è `long int`, l'altro è convertito in `long int`;
  D. altrimenti, se un operando è `unsigned int`, l'altro è convertito in `unsigned int`;
  E. altrimenti, entrambi gli operandi sono convertiti in `int`.

Esempi: conversione implicita

- Il tipo di dato `int x/y` è convertito in `double` prima di essere assegnato a `z`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = x/y; // z=1.0
```

Esempi: conversione implicita

- Il tipo di dato `int x/y` è convertito in `double` prima di essere assegnato a `z`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = x/y; // z=1.0
```

- Il tipi di dato `int x` e `y` sono convertiti in `double` per il calcolo dell'espressione `1.0*x/y`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = 1.0*x/y; // z=1.5
```

Esempi: conversione implicita

- Il tipo di dato `int x/y` è convertito in `double` prima di essere assegnato a `z`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = x/y; // z=1.0
```

- Il tipi di dato `int x` e `y` sono convertiti in `double` per il calcolo dell'espressione `1.0*x/y`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = 1.0*x/y; // z=1.5
```

- Il tipo di dato `int x` è convertito in `double` per il calcolo dell'espressione `x/y`.

```
1 int    x = 3;
2 double y = 2;
3 double z = x/y; // z=1.5
```

Esempi: conversione implicita

- Il tipo di dato `int x/y` è convertito in `double` prima di essere assegnato a `z`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = x/y; // z=1.0
```

- Il tipi di dato `int x` e `y` sono convertiti in `double` per il calcolo dell'espressione `1.0*x/y`.

```
1 int    x = 3;
2 int    y = 2;
3 double z = 1.0*x/y; // z=1.5
```

- Il tipo di dato `int x` è convertito in `double` per il calcolo dell'espressione `x/y`.

```
1 int    x = 3;
2 double y = 2;
3 double z = x/y; // z=1.5
```

- Il tipo di dato `double x/y` è convertito (e troncato) in `int` per essere assegnato a `z`.

```
1 int    x = 3;
2 double y = 2;
3 int    z = x/y; // z=1
```

Conversione di tipo implicita: assegnamento

- Nel caso di **assegnamento**, il tipo del risultato dell'espressione è convertito nel tipo della variabile a cui viene assegnato il risultato.
- L'assegnamento di un tipo di dato ad un tipo più *piccolo* può causare **troncamenti** o **overflow**.
- Esempio:

```
1 double x1 = DBL_MAX, x2 = 3.14;
2 int y1 = x1; // Overflow
3 int y2 = x2; // Arrotondamento: y2=3
```

Nell'esempio sopra il valore della variabile `y1` è non definito. Il valore di `y2` è troncato alla parte intera 3 di `3.14`.

Esempio: problemi con le conversioni implicite 1/2

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   unsigned int x = 1;
4   int y = -1;
5   printf("%d\n",x > y);
6   return 0;
7 }
```

Esempio: problemi con le conversioni implicite 1/2

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   unsigned int x = 1;
4   int y = -1;
5   printf("%d\n",x > y);
6   return 0;
7 }
```

Per la regola D di *integral promotion*, il valore **con segno** della variabile y viene convertito in unsigned int. In complemento a due, il numero -1 corrisponde alla configurazione di soli 1, che corrisponde a UINT_MAX, se il numero non ha segno. L'espressione relazionale x>y (riga 5) viene quindi valutata come 1>UINT_MAX $\Longrightarrow$ 0.

Esempio: problemi con le conversioni implicite 1/2

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   unsigned int x = 1;
4   int y = -1;
5   printf("%d\n",x > y);
6   return 0;
7 }
```

Per la regola D di *integral promotion*, il valore **con segno** della variabile y viene convertito in `unsigned int`. In complemento a due, il numero -1 corrisponde alla configurazione di soli 1, che corrisponde a `UINT_MAX`, se il numero non ha segno. L'espressione relazionale x>y (riga 5) viene quindi valutata come 1>`UINT_MAX` $\Longrightarrow$ 0.

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   unsigned int x = 1;
4   long int y = -1;
5   printf("%d\n",x > y);
6   return 0;
7 }
```

Esempio: problemi con le conversioni implicite 1/2

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   unsigned int x = 1;
4   int y = -1;
5   printf("%d\n",x > y);
6   return 0;
7 }
```

Per la regola D di *integral promotion*, il valore **con segno** della variabile y viene convertito in `unsigned int`. In complemento a due, il numero -1 corrisponde alla configurazione di soli 1, che corrisponde a `UINT_MAX`, se il numero non ha segno. L'espressione relazionale x>y (riga 5) viene quindi valutata come 1>`UINT_MAX` $\Longrightarrow$ 0.

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   unsigned int x = 1;
4   long int y = -1;
5   printf("%d\n",x > y);
6   return 0;
7 }
```

Dipende dalla dimensione assegnata dal compilatore ai tipi `long int` e `unsigned int`.

- Suggerimenti: evitare di mixare tipi interi con e senza segno in espressioni.

Esempio: problemi con le conversioni implicite 2/2

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   float  x = 0.1;
4   double y = 0.1;
5
6   printf("%d\n",x == y);
7   return 0;
8 }
```

Esempio: problemi con le conversioni implicite 2/2

- Cosa stampa il seguente codice?

```
1 #include <stdio.h>
2 int main() {
3   float  x = 0.1;
4   double y = 0.1;
5
6   printf("%d\n",x == y);
7   return 0;
8 }
```

Per la regola 2 delle *usual arithmetic conventions*, il contenuto di `x` viene convertito in `double`. Ricordiamo che 0.1 non ha una rappresentazione finita in binario: la rappresentazione di 0.1 in precisione singola (`float`) è meno accurata della sua rappresentazione in doppia precisione (`double`). La conversione di `x` in `double` non cambia la precisione della rappresentazione di `0.1` $\Longrightarrow$ la `printf()` stampa `0`.

Esempio: problemi con le conversioni implicite 2/2

- Cosa stampa il seguente codice?

```c
#include <stdio.h>
int main() {
  float  x = 0.1;
  double y = 0.1;

  printf("%d\n",x == y);
  return 0;
}
```

Per la regola 2 delle *usual arithmetic conventions*, il contenuto di `x` viene convertito in `double`. Ricordiamo che 0.1 non ha una rappresentazione finita in binario: la rappresentazione di 0.1 in precisione singola (`float`) è meno accurata della sua rappresentazione in doppia precisione (`double`). La conversione di `x` in `double` non cambia la precisione della rappresentazione di `0.1` $\Longrightarrow$ la `printf()` stampa `0`.

- Per ottenere il risultato *atteso* dobbiamo **forzare** esplicitamente una conversione di tipo da `double` a `float`.

```c
#include <stdio.h>
int main() {
  float  x = 0.1;
  double y = 0.1;
  printf("%d\n",x == (float)y);
  return 0;
}
```

Operatore di conversione esplicita di tipo: cast

- In qualsiasi espressione è possibile forzare **esplicitamente** la conversione di tipo (**type-casting**), utilizzando l'operatore (unario) di cast:

  `(<Tipo>) <Espressione>`

- Ad esempio, possiamo convertire un `int` in `double` con l'operatore di cast e *forzare* l'esecuzione della divisione tra numeri reali:

```
1 int x = 3;
2 int y = 2;
3 double z = (double)x/y; // z=1.5
```

Da notare che l'operatore di cast ha precedenza maggiore dell'operatore `/`, quindi il typecasting viene effettuato prima della divisione.

Operatore di conversione esplicita di tipo: cast

- In qualsiasi espressione è possibile forzare **esplicitamente** la conversione di tipo (**type-casting**), utilizzando l'operatore (unario) di cast:

  `(<Tipo>) <Espressione>`

- Ad esempio, possiamo convertire un `int` in `double` con l'operatore di cast e *forzare* l'esecuzione della divisione tra numeri reali:

```
1 int x = 3;
2 int y = 2;
3 double z = (double)x/y; // z=1.5
```

  Da notare che l'operatore di cast ha precedenza maggiore dell'operatore `/`, quindi il typecasting viene effettuato prima della divisione.

- Esempio: come *estrarre* la parte intera e frazionaria di un `double`

```
1 double pi        = 3.14;
2 int    int_part  = (int)pi;      // int_part = 3
3 double frac_part = pi-int_part; // frac_part = 0.14
```

Operatori logici

Operatore	Descrizione	Tipo	Esempio
`!`	Calcola il NOT logico dell'operando	Unario	`!x`
`&&`	Calcola l'AND logico tra i due operandi	Binario	`x && y`
`||`	Calcola l'OR logico tra i due operandi		`x || y`

- Gli **operatori logici** sono operatori **booleani**.
- Come gli operatori relazionali, generano un risultato **logico**: vero (1) o falso (0) di tipo `int`. Nota: in C il valore 0 indica falso e $\neq 0$ indica vero.

```c
int x = 3, y = 0;
int z = x && y; // z=0
```

```c
int x = 3, y = 0;
int z = x || y; // z=1
```

```c
int x = 3;
int z = !x; // z=0
```

Operatori logici

Operatore	Descrizione	Tipo	Esempio
`!`	Calcola il NOT logico dell'operando	Unario	`!x`
`&&`	Calcola l'AND logico tra i due operandi	Binario	`x && y`
`||`	Calcola l'OR logico tra i due operandi		`x || y`

- Gli **operatori logici** sono operatori **booleani**.
- Come gli operatori relazionali, generano un risultato **logico**: vero (1) o falso (0) di tipo `int`. Nota: in C il valore 0 indica falso e $\neq 0$ indica vero.

```c
int x = 3, y = 0;
int z = x && y; // z=0
```

```c
int x = 3, y = 0;
int z = x || y; // z=1
```

```c
int x = 3;
int z = !x; // z=0
```

- A differenza degli altri operatori visti finora, gli operatori logici hanno un ben preciso **ordine di valutazione**: da sinistra a destra (sequence point su && e ||).

```c
int x = 0, y = x++ || ++x;
// Definita! x=2, y=1
```

```c
int x = 0, y = ++x && x++;
// Definita! x=2, y=1
```

Operatori logici

Operatore	Descrizione	Tipo	Esempio
`!`	Calcola il NOT logico dell'operando	Unario	`!x`
`&&`	Calcola l'AND logico tra i due operandi	Binario	`x && y`
`||`	Calcola l'OR logico tra i due operandi		`x || y`

- Gli **operatori logici** sono operatori **booleani**.
- Come gli operatori relazionali, generano un risultato **logico**: vero (1) o falso (0) di tipo `int`. Nota: in C il valore 0 indica falso e $\neq 0$ indica vero.

```c
int x = 3, y = 0;
int z = x && y; // z=0
```

```c
int x = 3, y = 0;
int z = x || y; // z=1
```

```c
int x = 3;
int z = !x; // z=0
```

- A differenza degli altri operatori visti finora, gli operatori logici hanno un ben preciso **ordine di valutazione**: da sinistra a destra (sequence point su && e ||).

```c
int x = 0, y = x++ || ++x;
// Definita! x=2, y=1
```

```c
int x = 0, y = ++x && x++;
// Definita! x=2, y=1
```

- Ulteriore regola: se il primo operando è sufficiente a determinare il valore dell'espressione, il secondo non è valutato (**short-circuit evaluation**).

```c
int x = 3, y = 0;
int z = y && ++x;
// z=0, x=3
```

```c
int x = 3, y = 0;
int z = x || y++;
// z=1, y = 0
```

```c
int x = 3, y=0;
int z = y++ || x++;
// z=1, y=1, x=4
```

Operatori logici: tabelle

AND	T	F
T	T	F
F	F	F

OR	T	F
T	T	T
F	T	F

NOT	T	F
	F	T

T = TRUE, **F** = FALSE

Operatore condizionale

- Le **espressioni condizionali** fanno uso dell'**operatore condizionale** (terziario)

  `<Espressione1> ? <Espressione2> : <Espressione3>`

- L'operatore condizionale ha un ben preciso **ordine di valutazione** da sinistra a destra.
- Viene prima valutata l'`Espressione1`. Se questa è vera (ovvero, ha valore diverso da 0) allora viene valutata l'`Espressione2`, altrimenti si valuta l'`Espressione3`. Il valore della seconda valutazione diventa il valore dell'intero costrutto.

Operatore condizionale

- Le **espressioni condizionali** fanno uso dell'**operatore condizionale** (terziario)

  `<Espressione1> ? <Espressione2> : <Espressione3>`

- L'operatore condizionale ha un ben preciso **ordine di valutazione** da sinistra a destra.
- Viene prima valutata l'`Espressione1`. Se questa è vera (ovvero, ha valore diverso da 0) allora viene valutata l'`Espressione2`, altrimenti si valuta l'`Espressione3`. Il valore della seconda valutazione diventa il valore dell'intero costrutto.
- Se uno dei due rami non viene scelto, l'espressione corrispondente non viene valutata (vedi terzo esempio).

```c
int x = 3, y = 0;
int z = x==y ? x : y;
// z = y = 0, x = 3
```

```c
int x = 3, y = 0;
int z = y<=x ? y++ : 1;
// z=0, y=1, x = 3
```

```c
int x = 3, y=0;
int z = x!=y? ++y : ++x;
// z = y = 1, x = 3
```

Operatore condizionale

- Le **espressioni condizionali** fanno uso dell'**operatore condizionale** (terziario)

  `<Espressione1> ? <Espressione2> : <Espressione3>`

- L'operatore condizionale ha un ben preciso **ordine di valutazione** da sinistra a destra.
- Viene prima valutata l'`Espressione1`. Se questa è vera (ovvero, ha valore diverso da 0) allora viene valutata l'`Espressione2`, altrimenti si valuta l'`Espressione3`. Il valore della seconda valutazione diventa il valore dell'intero costrutto.
- Se uno dei due rami non viene scelto, l'espressione corrispondente non viene valutata (vedi terzo esempio).

```c
int x = 3, y = 0;
int z = x==y ? x : y;
// z = y = 0, x = 3
```

```c
int x = 3, y = 0;
int z = y<=x ? y++ : 1;
// z=0, y=1, x = 3
```

```c
int x = 3, y=0;
int z = x!=y? ++y : ++x;
// z = y = 1, x = 3
```

- L'operatore condizionale ha un sequence point sul `?`, subito dopo `Espressione1`.

```c
int x = 0, y;
y = ++x ? x-- : x++;
// Definita! x=0, y=1
```

```c
int x = 0;
x = ++x ? x-- : x++;
// Non definita! -> x = x--
```

Operatori bitwise

Operatore	Descrizione	Tipo	Esempio
~	Calcola il complemento ad uno dell'operando	Unario	~x
&	Calcola l'AND logico bit a bit tra i due operandi	Binario	x & y
|	Calcola l'OR logico bit a bit tra i due operandi		x | y
^	Calcola lo XOR logico bit a bit tra i due operandi		x ^ y
«	Calcola la shift a sinistra del primo operando per un numero di bit pari al valore del secondo operando		x « y
»	Calcola la shift a destra del primo operando per un numero di bit pari al valore del secondo operando		x » y

- Gli **operatori bitwise** permettono di effettuare operazioni logiche a livello di rappresentazione binaria dei dati.

```
1  // Assumiano che il tipo char sia signed (sizeof(char)=1)
2
3  char x = 97; // x = 'a'      01100001
4  char y = 65; // y = 'A'      01000001
5  char z;
6
7  z = ~x;      // z = -98      10011110
8  z = ~x+1;    // z = -97      10011111  Complemento a 2
9  z = x | y;   // z = x = 97   01100001
10 z = x & y;   // z = y = 65   01000001
11 z = x ^ y;   // z = 32       00100000
12 z = x << 1;  // z = 194      11000010
13 z = y >> 2;  // z = 16       00010000
```

- Possono essere utilizzati solo con tipi di dato intero.

Esempio: stampa della codifica binaria di un char

```
1  #include<stdio.h>
2  // Stampa la codifica binaria del carattere 'a'
3  // Ricordiamo che sizeof(char)=1
4  int main() {
5   char x = 'a'; // Equivalente a 97
6
7   x & (1<<7) ? putchar('1') : putchar('0');
8   x & (1<<6) ? putchar('1') : putchar('0');
9   x & (1<<5) ? putchar('1') : putchar('0');
10  x & (1<<4) ? putchar('1') : putchar('0');
11  x & (1<<3) ? putchar('1') : putchar('0');
12  x & (1<<2) ? putchar('1') : putchar('0');
13  x & (1<<1) ? putchar('1') : putchar('0');
14  x & (1<<0) ? putchar('1') : putchar('0');
15
16  putchar('\n');
17  return 0;
18 }
```

- Se `x` è un `char`, `putchar(x)` è equivalente a `printf("%c",x)`.
- L'istruzione `1 << i` sposta l'unico bit *acceso* nella costante `int 1` nella posizione `7-i`, con `i` che varia da 0 a 7.
- L'operatore bitwise `&` mette a zero tutti i bit nella variabile `x`, tranne eventualmente il bit nella posizione indicizzata con `i`.
- Il valore dell'espressione `x & (1<<i)` è 1 solo se `x` ha un bit *acceso* in posizione `i`.

Operatori di assegnamento

Operatore	Descrizione	Tipo	Esempio
=	Assegna il valore del secondo operando al primo operando	Semplice	x = y
+=	Assegna al primo operando la somma tra il primo e secondo operando	Composto	x += y
-=	Assegna al primo operando la differenza tra il primo e secondo operando		x -= y
/=	Assegna al primo operando il risultato della divisione tra primo e secondo operando		x /= y
*=	Assegna al primo operando il risultato della moltiplicazione tra primo e secondo operando		x *= y
%=	Assegna al primo operando il risultato della modulo tra primo e secondo operando		x %= y
^=	Assegna al primo operando il risultato dello XOR bitwise tra primo e secondo operando		x ^= y
|=	Assegna al primo operando il risultato dello OR bitwise tra primo e secondo operando		x |= y
&=	Assegna al primo operando il risultato dello AND bitwise tra primo e secondo operando		x &= y
«=	Assegna al primo operando il risultato dello shift a sinistra		x «= y
»=	Assegna al primo operando il risultato dello shift a destra		x »= y

- L'operatore di assegnamento è utilizzato per memorizzare dei valori in una variabile.

<Nome Variabile> = <Espressione>

- L'operatore a sinistra non può essere una costante o una espressione composta.

```
1 1   = x+y;// Errore: costante a sinistra
2 x+1 = 10; // Errore: espressione composta a sinistra
3 -x  = y;  // Errore: espressione composta a sinistra
4 ++x = y;  // Errore: espressione composta a sinistra
```

Operatori di assegnamento

Operatore	Descrizione	Tipo	Esempio
=	Assegna il valore del secondo operando al primo operando	Semplice	x = y
+=	Assegna al primo operando la somma tra il primo e secondo operando	Composto	x += y
-=	Assegna al primo operando la differenza tra il primo e secondo operando		x -= y
/=	Assegna al primo operando il risultato della divisione tra primo e secondo operando		x /= y
*=	Assegna al primo operando il risultato della moltiplicazione tra primo e secondo operando		x *= y
%=	Assegna al primo operando il risultato della modulo tra primo e secondo operando		x %= y
^=	Assegna al primo operando il risultato dello XOR bitwise tra primo e secondo operando		x ^= y
|=	Assegna al primo operando il risultato dello OR bitwise tra primo e secondo operando		x |= y
&=	Assegna al primo operando il risultato dello AND bitwise tra primo e secondo operando		x &= y
«=	Assegna al primo operando il risultato dello shift a sinistra		x «= y
»=	Assegna al primo operando il risultato dello shift a destra		x »= y

- L'operatore di assegnamento è utilizzato per memorizzare dei valori in una variabile.

<Nome Variabile> = <Espressione>

- L'operatore a sinistra non può essere una costante o una espressione composta.

```
1 1   = x+y; // Errore: costante a sinistra
2 x+1 = 10;  // Errore: espressione composta a sinistra
3 -x  = y;   // Errore: espressione composta a sinistra
4 ++x = y;   // Errore: espressione composta a sinistra
```

- Una assegnazione è considerata una espressione, con un proprio risultato. E' quindi possibile eseguire assegnamenti multipli in una sola riga di codice.

```
1 int x, y = 1, z = 0;
2
3 z = y = (x   = y + 2) - 1;
4 // z = y = 2, x = 3
```

$\Longrightarrow$

```
1 int x, y = 1, z = 0;
2 x = y + 2; // x = 3
3 y = x - 1; // y = 2
4 z = y;     // z = 2
```

Operatori di assegnamento: ulteriori esempi

- Qual è il risultato dei seguenti assegnamenti sulla variabile y?

```c
int x = 0, y;
y = (x=1) && x;
```

```c
int x = 0, y;
y = (x=1) * x;
```

Operatori di assegnamento: ulteriori esempi

- Qual è il risultato dei seguenti assegnamenti sulla variabile y?

```
1 int x = 0, y;
2 y = (x=1) && x;
```

```
1 int x = 0, y;
2 y = (x=1) * x;
```

Risposta: y=1 nel primo esempio e non definito nel secondo esempio. Il risultato nel secondo esempio dipende dall'ordine di valutazione degli operandi di $*$. Viola la regola 2 introdotta precedentemente.

Operatori di assegnamento: ulteriori esempi

- Qual è il risultato dei seguenti assegnamenti sulla variabile y?

```
1 int x = 0, y;
2 y = (x=1) && x;
```

```
1 int x = 0, y;
2 y = (x=1) * x;
```

Risposta: y=1 nel primo esempio e non definito nel secondo esempio. Il risultato nel secondo esempio dipende dall'ordine di valutazione degli operandi di *. Viola la regola 2 introdotta precedentemente.

- Il seguente codice è valido?

```
1 int x, y;
2 x = 0 = y;
```

```
1 int x, y;
2 (x = 0) = y;
```

Operatori di assegnamento: ulteriori esempi

- Qual è il risultato dei seguenti assegnamenti sulla variabile `y`?

```c
int x = 0, y;
y = (x=1) && x;
```

```c
int x = 0, y;
y = (x=1) * x;
```

Risposta: `y=1` nel primo esempio e non definito nel secondo esempio. Il risultato nel secondo esempio dipende dall'ordine di valutazione degli operandi di `*`. Viola la regola 2 introdotta precedentemente.

- Il seguente codice è valido?

```c
int x, y;
x = 0 = y;
```

```c
int x, y;
(x = 0) = y;
```

Risposta: Errore di sintassi in entrambi i casi. L'operatore di assegnamento è associativo da destra verso sinistra. Nel primo esempio cerchiamo di assegnare un valore alla costante intera 0 (`0 = y`). Nel secondo esempio cerchiamo di assegnare un valore al risultato dell'espressione `x = 0` (anche in questo caso `0 = y`).

Operatori di assegnamento composto

- Gli operatori di assegnamento composto permettono di scrivere in forma compatta espressioni che usano il vecchio valore per calcolare il nuovo.
- La sintassi generale degli operatori composti è la seguente

  `<Nome Variabile> <Operatore>= <Espressione>`

  dove `Operatore` è uno degli operatori binari `+`, `-`, `/`, `*`, `%`, `&`, `«`, `»`, `|`, `^`

Operatori di assegnamento composto

- Gli operatori di assegnamento composto permettono di scrivere in forma compatta espressioni che usano il vecchio valore per calcolare il nuovo.
- La sintassi generale degli operatori composti è la seguente

  `<Nome Variabile> <Operatore>= <Espressione>`

  dove `Operatore` è uno degli operatori binari `+`, `-`, `/`, `*`, `%`, `&`, `«`, `»`, `|`, `^`

```
1 int x, y, z;
2 x = y = z = 2;
3
4 x += 3;       // x = 5
5 y *= x + 5;   // y = 20
6 z /= x + y;   // z = 0
```

$\Longrightarrow$

```
1 int x, y, z;
2 x = y = z = 2;
3
4 x = x + 3;         // x = 5
5 y = y * (x + 5);   // y = 20
6 z = z / (x + y);   // z = 0
```

Operatori di assegnamento composto

- Gli operatori di assegnamento composto permettono di scrivere in forma compatta espressioni che usano il vecchio valore per calcolare il nuovo.
- La sintassi generale degli operatori composti è la seguente

  `<Nome Variabile> <Operatore>= <Espressione>`

  dove `Operatore` è uno degli operatori binari `+`, `-`, `/`, `*`, `%`, `&`, `«`, `»`, `|`, `^`

```
1 int x, y, z;
2 x = y = z = 2;
3
4 x += 3;       // x = 5
5 y *= x + 5;   // y = 20
6 z /= x + y;   // z = 0
```

$\Longrightarrow$

```
1 int x, y, z;
2 x = y = z = 2;
3
4 x = x + 3;         // x = 5
5 y = y * (x + 5);   // y = 20
6 z = z / (x + y);   // z = 0
```

- Anche con gli operatori composti è possibile definire assegnamenti multipli.

```
1 int x = 3, y = 1, z = 1;
2
3 z += y /= (x *= y + 2) - 1;
4 // z = 1, y = 0, x = 9
```

$\Longrightarrow$

```
1 int x = 3, y = 1, z=1;
2
3 x = x * (y + 2);   // x = 9
4 y = y / (x - 1);   // y = 0
5 z = z + y;         // z = 1
```

Operatore di concatenazione (virgola)

- L'**operatore di concatenzazione** o **operatore virgola** (binario) consente di unire due espressioni.

`<Espressione1>, <Espressione2>`

- Il primo operando dell'operatore virgola (`Espressione1`) viene valutato e il suo valore è ignorato.
- Il secondo operando (`Espressione2`) viene valutato e il suo valore restituito come risultato dell'espressione.
- L'ordine di valutazione dell'operatore virgola è ben definito da sinistra a destra.
- Ha un sequence point sulla virgola, dopo la prima espressione `Espressione1`.

```
1 int x, y=0;
2
3 x = (y = 2, y + 2);
4 // x = 4, y = 2
```

```
1 int x, y=0;
2
3 x = (y++, y--);
4 // Definita! x = 1, y = 0
```

Operatore di concatenazione (virgola)

- L'**operatore di concatenzazione** o **operatore virgola** (binario) consente di unire due espressioni.

`<Espressione1>, <Espressione2>`

- Il primo operando dell'operatore virgola (`Espressione1`) viene valutato e il suo valore è ignorato.
- Il secondo operando (`Espressione2`) viene valutato e il suo valore restituito come risultato dell'espressione.
- L'ordine di valutazione dell'operatore virgola è ben definito da sinistra a destra.
- Ha un sequence point sulla virgola, dopo la prima espressione `Espressione1`.

```c
int x, y=0;

x = (y = 2, y + 2);
// x = 4, y = 2
```

```c
int x, y=0;

x = (y++, y--);
// Definita! x = 1, y = 0
```

- Negli esempi sopra le parentesi sono rilevanti dato che l'operatore virgola ha la più bassa priorità tra gli operatori C.

```c
int x, y=0;

x = y = 2, y + 2;
// x = 2, y = 2
```

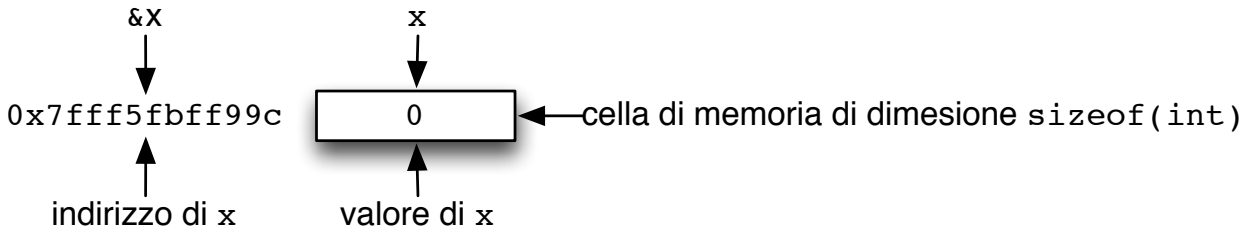
```c
int x, y=0;

x = y++, y--;
// x = 0, y = 0
```

Operatore di indirizzo (brevissima introduzione)

- L'operatore di indirizzo &, restituisce l'indirizzo di memoria di una variabile.

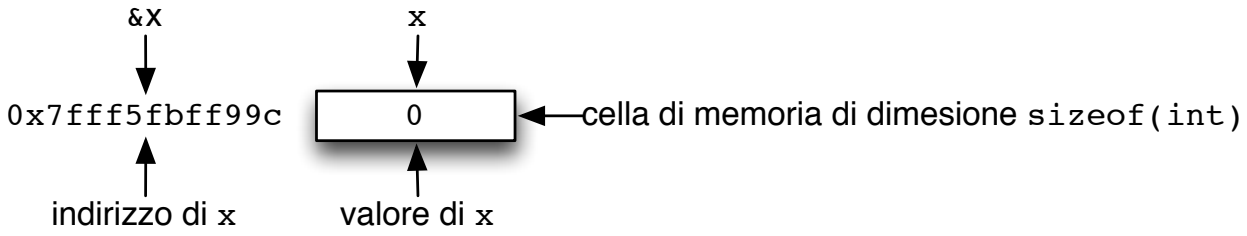
```
1 int x=0;
2 printf("Indirizzo di x = %p, Valore di x = %d\n",&x,x);
```



Operatore di indirizzo (brevissima introduzione)

- L'operatore di indirizzo &, restituisce l'indirizzo di memoria di una variabile.

```
1 int x=0;
2 printf("Indirizzo di x = %p, Valore di x = %d\n",&x,x);
```



- Al momento, l'operatore & ci serve solo per introdurre la funzione di libreria `scanf()`.
- La funzione `scanf()`, simmetricamente alla `printf()`, ci permette di leggere *input formattato* da tastiera (approfondimento in laboratorio).
- La `scanf()` richiede che gli argomenti siano *indirizzi di memoria* di variabili (vedi riga 6) e *non valori* di variabili.

```
1 #include <stdio.h>
2
3 int main() {
4   int x;
5   printf("Inserisci un intero: ");
6   scanf("%d",&x);
7   printf("Hai inserito l'intero %d\n",x);
8   return 0;
9 }
```

La libreria `math.h`

- Gli operatori del linguaggio C non ci permettono di effettuare calcoli *complessi*, come ad esempio il calcolo della radice quadrata di un numero.
- Le funzioni matematiche più comuni sono fornite come *funzioni di libreria* standard, specificate nell'header `math.h`.
- L'header `math.h` specifica una sola macro e un certo numero di funzioni.
- Le funzioni matematiche prendono in input tipi di dato `double` e ritornano tipi di dato `double`.
- L'unica macro presente in `math.h` è `HUGE_VAL`, che rappresenta un valore troppo grande per essere rappresentabile in virgola mobile.
- Viene utilizzato come valore di ritorno delle funzioni (con segno positivo o negativo), quando il risultato di una operazione matematica genera un valore troppo grande da rappresentare.
- *Approssimativamente*, `HUGE_VAL` può essere interpretato come *infinito* e `-HUGE_VAL` come *meno infinito*.

Funzioni della libreria `math.h`

Classe	Funzione	Descrizione
Funzioni trigonometriche	`double acos(double x);`	**arcocoseno**
	`double asin(double x);`	**arcoseno**
	`double atan(double x);`	**arcotangente**
	`double atan2(doubly y, double x);`	**arcotangente di due parametri**
	`double cos(double x);`	**coseno**
	`double cosh(double x);`	**coseno iperbolico**
	`double sin(double x);`	**seno**
	`double sinh(double x);`	**seno iperbolico**
	`double tan(double x);`	**tangente**
	`double tanh(double x);`	**tangente iperbolica**
Funzioni esponenziali e logaritmiche	`double exp(double x)`	**esponenziale**
	`double frexp(double value, int *exp);`	**funzione a potenza di due**
	`double ldexp(double x, int exp);`	**operazione in virgola mobile**
	`double log(double x);`	**logaritmo naturale**
	`double log10(double x);`	**logaritmo in base 10**
	`double modf(double value, double *iptr);`	**estrae la parte frazionaria e intera di** x
Funzioni potenza	`double pow(double x, double y);`	**elevamento a potenza**
	`double sqrt(double x);`	**radice quadrata**
Funzioni di arrotondamento, valore assoluto e resto	`double ceil(double x);`	**più piccolo intero non minore di** x
	`double fabs(double x);`	**valore assoluto**
	`double floor(double x);`	**più grande intero non maggiore di** x
	`double fmod(double x, double y);`	**resto del numero in virgola mobile**