Espressioni
e
Conversioni di tipo

Espressioni in C

Espressioni semplici

- una costante è un' espressione
- una variabile è un' espressione

Gli **operatori** vengono usati per costruire espressioni a partire da espressioni fornite in input come argomenti.

Il C fornisce un' ampia varietà di operatori.

Non solo
•operatori aritmetici
•operatori relazionali
•operatori logici

… ma anche
•l' operatore di assegnamento
•operatori di incremento e decremento
e tanti altri…

Espressioni in C (cont.)

- Ogni espressione **denota un valore**
  - Per valutare espressioni è necessario conoscere le proprietà di **associatività** e **priorità** degli operatori
- Espressioni di origine diversa (aritmetiche e booleane) e di tipo diverso possono essere combinate in un'unica espressione

Ma attenzione alla **compatibilità di tipo**!

Esempio

- (5.0+7) + (x && y) è legale
- % si applica solo a operandi di tipo `int`. Quindi
  - 'a' %(x>7) è legale
  - (7.5+2.9)%3 è illegale
- 7.5 + 6 è legale

➔In caso di incompatibilità di tipo è necessario applicare delle **Conversioni di tipo**

Operatore di assegnamento

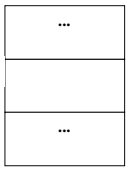
- Denotato mediante il simbolo **=**

  (l' operatore relazionale di uguaglianza è denotata con il simbolo **==**)
- Viene utilizzato per assegnare ad una variabile il valore di un' espressione

Esempio

```
int  N;
```

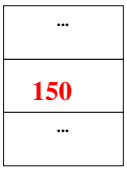
simbolo	indirizzo
N	1600



L' esecuzione di una **dichiarazione** provoca l' allocazione di uno spazio in memoria equivalente 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 **=**

Operatore di assegnamento (*cont.*)

- In C l'assegnamento non è un'istruzione ma un **operatore**.
- Quindi può essere utilizzato nelle espressioni assieme ad *altri operatori* e *può comparire più volte*
- L'esecuzione di **un'operazione di assegnamento (v=e)** comporta innanzitutto la valutazione dell'espressione (a destra dell'assegnamento)
- Dopodiché, si inserisce il valore risultante nella locazione di memoria relativa alla variabile (posta a sinistra dell'assegnamento)

➔L'**associatività** dell'operatore di assegnamento è **da destra a sinistra**

Esempio

```
c = 2;
d = (c+5)/3 - c;
d = (d+c)/2;
```


Proprietà dell'assegnamento e Inizializzazione

- L'operazione di assegnamento è una espressione il cui valore è il valore assegnato all'operando di sinistra

Esempi

```
k = j = 5;
```

equivale a

```
j = 5;
k = j;
```

~~`k = j+2 = 5;`~~

```
k = (x=8)||y && z;
```

- Il primo assegnamento di un valore ad una variabile dichiarata viene detto **inizializzazione**.

L'inizializzazione <u>si può</u> effettuare anche al momento della dichiarazione.

Esempio `int a, b=56;`

Operatori di incremento e decremento (++ e --)

++ aggiunge 1 al suo operando: **a = a+1;**
è equivalente a **a++;** e a **++a;**

– – sottrae 1 al suo operando: **a = a-1;**
è equivalente a **a--;** e a **--a;**

- Questi operatori possono essere usati nelle espressioni dove possono comparire più volte
- Nelle espressioni sono utilizzabili in due modi:
  - **come pre-operatori (*notazione prefissa*)**
    - **++var** *prima* incremento, *poi* utilizzo (pre-incremento)
    - **– –var** *prima* decremento, *poi* utilizzo (pre-decremento)
  - **come post-operatori (*notazione postfissa*)**
    - **var++** *prima* utilizzo, *poi* incremento (post-incremento)
    - **var– –** *prima* utilizzo, *poi* decremento (post-decremento)

La differenza tra le notazioni si nota nelle espressioni:
b = a++; è diverso da **b = ++a;**

Operatori di incremento e decremento (*cont.*) (++ e --)

Esempi

```
int a,b;
a=5;
b= a++;          /* a==6;  b==5 */
a=5;
b= ++a;          /* a==6;  b==6 */

int i, k = 5;
i = ++k;         /*  i = 6, k = 6  */
i = k++;         /*  i = 6, k = 7  */

int i=4, j, k = 5;
j = i + k++;  /*  i = 4, j = 9, k = 6  */
```

Ricordarsi di risolvere questi esercizi "passo per passo", rappresentando i contenuti delle locazioni di memoria corrispondenti alle variabili

Assegnamenti "compatti"

Gli **operatori composti** consentono di abbreviare assegnamenti che utilizzano il vecchio valore di una variabile per calcolarne quello nuovo :

a = a+b;	a+ =b;
a = a-b;	a- =b;
a = a*b;	a*=b;
a = a/b;	a/=b;
a = a%b;	a%=b;

Priorità e associatività degli operatori

PRIORITA' : specifica l'ordine di valutazione degli operatori quando in una espressione compaiono *operatori diversi*

Esempio **3 + 10 * 20** è letta come **3 + (10 * 20)** perché in C l'operatore * è <u>prioritario</u> rispetto a +

Nota operatori diversi possono comunque avere uguale priorità (per esempio, + e -, * e / e %)

ASSOCIATIVITA' : precisa l'ordine di valutazione quando in un'espressione compaiono *operatori con la medesima priorità*

associatività a sinistra → valutazione da sinistra a destra

associatività a destra → valutazione da destra a sinistra

Esempio **30 - 10 + 8** è letta come **(30 – 10) + 8**, perché gli operatori + e – sono ***equiprioritari***, e sono entrambi ***associativi a sinistra***

Priorità e associatività degli operatori (*cont.*)

Le espressioni aritmetiche vengono valutate in base alle regole tradizionali di priorità degli operatori in aritmetica.

Esempio

a=2, b=4, c=6,

a+b*c → a+(b*c) = 26

Associatività e priorità possono venire alterate mediante l'uso di parentesi:

a+b*c = 26 (a+b)*c = 36

a-b+c = 4 a-(b+c) = -8

2-3+4*7%2+3 = 2 2-3+4*(7%(2+3)) = 7

Priorità e associatività degli operatori (*cont.*)

Gli operatori relazionali hanno priorità inferiore agli operatori aritmetici.

k<b+3 equivale a k<(b+3) e non a (k<b) +3

Esempio *Cosa contengono le variabili logico1 e logico2 al termine dell'esecuzione del seguente programma?*

```
main()
{ float x, y;
  int a,b;
  int logico1, logico2 ;
  x=10.5;  y=3.6;
  a=2;        b=1;
  logico1=x<y*0.5;
  logico2=a!=b+6;          /* logico1  ==  10.5<1.8 == 0 == FALSE */
}                          /* logico2  ==   2!=7      == 1 == TRUE */
```

Tabella di priorità e associatività*

pr. max

Operatore	Associatività
( ) [ ] -> .	da sn a ds
! - (unario) ++ --	da ds a sn
* / %	da sn a ds
+ -	da sn a ds
> >= < <=	da sn a ds
== !=	da sn a ds
&&	da sn a ds
||	da sn a ds
?:	da ds a sn
= += -= *= /= %=	da ds a sn

pr. min

(Gli operatori sulla stessa riga hanno stessa priorità)

* Fonte Kernighan Ritchie

Dove si sbaglia frequentemente …

Operazioni matematiche e tipi di dato

- Divisione fra interi e divisione fra reali (stesso simbolo /, ma differente significato)
- Significato e uso dell'operazione di modulo (%)
- Operatore di assegnamento (=) e operatore di uguaglianza (==)
- Notazione prefissa e postfissa di ++ e - - negli assegnamenti

Conversioni di tipo

- Per eseguire un'operazione aritmetica gli operandi devono essere delle stesse dimensioni (lo stesso numero di bit) e devono poter essere memorizzati nello stesso modo

Conversione implicita

- Quando **operandi di tipo diverso** vengono usati nelle stessa espressione, il compilatore C deve generare delle istruzioni che **convertono il tipo** di alcuni operandi in modo che l'hardware sia in grado di valutare l'espressione.

Esempi

- Uno `short` a 16-bit in una stessa espressione con un `int` a 32-bit: Il compilatore farà in modo che il valore `short` venga convertito a 32 bit.
- Se si aggiunge un `int` a un `float`, il compilatore provvederà a convertire l'`int` in formato `float`.

Conversioni implicite

Le conversioni implicite sono eseguite

- **Nelle espressioni:** quando gli operandi non sono dello stesso tipo (**conversioni aritmetiche normali**)
- **Negli assegnamenti (v=e):** quando il tipo di **e** non corrisponde al tipo di **v**
- **Nelle funzioni:**
  - quando il tipo dei parametri formali non corrisponde al tipo dei paramenti attuali
  - quando il tipo dell'espressione in `return` non corrisponde al tipo di ritorno della fuzione

Conversioni aritmetiche normali

In caso di operandi di tipo diverso

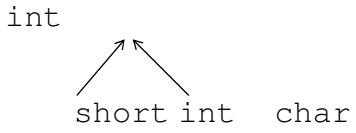
`op1 op op2  type(op1) ≠ type(op2)`

il C concilia con sicurezza entrambi i valori convertendo l'operando di tipo più piccolo nel tipo dell'altro operando (**promozione**)

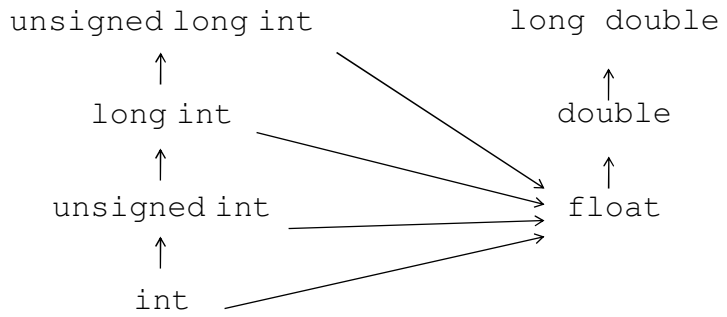
Esempio

- `f + i` con `f` di tipo `float` e `i` di tipo `int`, `i` viene convertito nel tipo `float`

Promozioni integrali



Conversioni aritmetiche normali



Attenzione! `int` ⟶ `unsigned int`

Esempio Nell' espressione `i<u` se `i` è di tipo `int` e `u` è di tipo `unsigned`, `i` viene convertito in `unsigned`.

Se i=-10 e u = 10 il risultato sarà 0 (false).

Spesso in questi casi il compilatore lancia un **Warning**

Conversioni negli assegnamenti

- Negli assegnamenti **v = e** : **e** viene convertita nel tipo di **v**

Esempi

```
char c;
int i;
float f;
double d;

i = c;    /* c viene convertita in int    */
f = i;    /* i viene convertita in float */
d = f;    /* f viene convertita in double */
```

Attenzione alle conversioni "inverse"

```
i = 842.97;     /* i vale 842 */
i = -842.97;    /* i vale -842 */
```

Conversioni esplicite

- Il C consente al programmatore di effettuare delle conversioni esplicite attraverso l'operazione di **casting**
- Una espressione di cast ha la forma

  (*type-name*) *expression*

  dove *type-name* specifica il tipo nel quale verrà convertita l'espressione

Esempio

```
float f, frac_part;
frac_part = f - (int) f;
```

La differenza tra `f` e `(int) f` è la **parte decimale** di `f`

Casting

Le conversioni di tipo possono essere usate per

- Documentare conversioni che accadrebbero comunque

Esempio `i = (int) f;`

- Forzare il compilatore ad effettuare le conversioni che vogliamo

Esempio

```
float quotient;
int dividend, divisor;
quotient = dividend / divisor;
```

Il risultato della divisione tra interi è un intero.
Per **evitare troncamenti**:

```
quotient = (float) dividend / divisor;
```

Nota basta convertire uno dei due operanti

Esempio

`(7.5+2.9)%3` è illegale: il compilatore segnala un errore!
`(int)(7.5+2.9)%3` è legale

Casting (cont.)

- (*type-name*) è un **operatore unario**
- Gli operatori unari hanno precedenza sugli operatori binari. Quindi

  `(float) dividend / divisor`

  viene interpretato come

  `((float) dividend) / divisor`
- Per effettuare il casting di una intera espressione `e`

  `(type name) (e)`

Attenzione

- `(float) dividend / divisor` ≠ `(float)(dividend/divisor)`
- `(float) (dividend/divisor)` causa troncamenti!