2.4 Espressioni

2.4.1 Espressioni condizionali

Si definisce *espressione aritmetica* un insieme di variabili, costanti e richiami di funzione connessi da operatori aritmetici. Il risultato di un'espressione aritmetica è sempre un valore numerico. La Figura 2.1 mostra gli operatori aritmetici e le loro priorità in ordine dalla più alta alla più bassa. Da osservare come non sia presente l'operatore di elevamento a potenza.

```
negazione (-unario)
 moltiplicazione (*), divisione (/), modulo (%)
somma (+), sottrazione (-)
= (assegnamento)
```

Figura 2.1 Gerarchia degli operatori aritmetici e di assegnamento

Quello di negazione è l'unico operatore unario, cioè che si applica a un solo operando. Se `x` ha valore 5, l'espressione

```
-x;
```

restituisce –5, mentre

```
2 * -(x-6);
```

restituisce 2. Si noti che mentre il – anteposto alla parentesi tonda aperta corrisponde all'operatore unario di negazione, l'altro rappresenta l'operatore binario di sottrazione.

L'operatore di modulo, `%`, consente di ottenere il resto della divisione intera tra l'operando che lo precede e quello che lo segue. Quindi, sempre nell'ipotesi che `x` valga 5,

```
34 % x;
```

ha valore 4, perché (34 : 5 = 6 con resto 4).

All'interno delle espressioni aritmetiche, la priorità degli operatori segue le regole dell'algebra. La valutazione di una espressione contenente operazioni matematiche avviene esaminandola da sinistra a destra più volte, dunque gli operatori sono associativi da sinistra verso destra. Tutte le operazioni di negazione sono eseguite per prime, quindi l'espressione è esaminata nuovamente per eseguire tutte le moltiplicazioni, divisioni e le operazioni modulo. Infine l'espressione viene sottoposta a scansione ancora una volta per eseguire le addizioni e le sottrazioni. La priorità degli operatori può essere alterata mediante le parentesi tonde: vengono valutate per prime le operazioni all'interno delle parentesi tonde più interne.

Osserviamo le seguenti espressioni, nell'ipotesi che le variabili intere `a`, `b` e `c` abbiano rispettivamente valore: 7, 3 e 5. Il risultato di

```
a + b - 15 + 4 * c
```

è 15, mentre il risultato di

```
a + b + c % 2
```

è 11, in quanto l'operatore modulo restituisce il resto della divisione intera tra il valore di `c` (5) e 2, ossia 1. Il risultato di

```
(a+b) * 32 + 4 * c
```

è 340 mentre quello di

```
((((((c + 6) * 3 + a) / 2) + 10 * 4) / 12) + b)
```

è 8. Non esiste alcun limite al numero di coppie di parentesi tonde impiegate.

L'assegnamento = è anch'esso un operatore, e ha quindi la sua posizione all'interno della scala di priorità (Figura 2.1); la sua priorità è minore di quella di tutti gli altri; per questa ragione nell'espressione

```
y = z * 2 / x
```

prima viene valutata completamente la parte a destra dell'assegnamento e poi il risultato viene immesso nella variabile `y`. L'operazione di assegnamento può essere multipla, per esempio:

```
x = y = z
```

In questo caso il valore di `z` viene assegnato a `y` e a `x`. Analogamente si può avere

```
a = b = c = f = d
```

dove il valore di `d` viene assegnato ad `a`, `b`, `c` e `f`, o anche

```
x = y = t = z*2/x;
```

dove il valore restituito da `z*2/x` viene assegnato a `x`, `y` e `t`.

2.4.2 Espressioni logiche

Un'*espressione logica* è un'espressione che genera come risultato un valore vero o falso (abbiamo visto che in C non esiste il tipo booleano, presente in altri linguaggi), e viene utilizzata dalle istruzioni di controllo. Le espressioni logiche, per la precisione, producono come risultato 1 per vero e 0 per falso (qualsiasi valore numerico diverso da zero viene comunque considerato vero). Un semplice esempio di espressione logica è una variabile il cui contenuto può essere interpretato in due modi: vero se diverso da zero, falso se uguale a zero.

Le espressioni logiche possono contenere gli *operatori relazionali*, usati per confrontare fra loro dei valori, riportati in Figura 2.2.

```
> (maggiore di)   >= (maggiore uguale)
< (minore di)     <= minore uguale)
```

```
== (uguaglianza) != (disuguaglianza)
```

Figura 2.2 Gerarchia degli operatori relazionali

Si noti come l'operatore di uguaglianza == sia diverso anche nella notazione da quello di assegnamento =, fatto a nostro avviso positivo e non comune negli altri linguaggi di programmazione.

La priorità di `>`, `>=`, `<`, e `<=` è la stessa ed è maggiore di == e `!=`. Dunque in

```
x>y == z>t
```

viene valutato prima `x>y` e `z>t` e successivamente verificata l'uguaglianza tra i due risultati come se l'espressione fosse: `(x>y)==(z>t)`.

```
!  (NOT logico)

&& (AND logico)

|| (OR logico)
```

Figura 2.3 Gerarchia degli operatori logici

Gli *operatori logici* consentono invece di concatenare fra di loro più espressioni logiche e di negare il risultato di un'espressione logica; essi hanno la scala di priorità di Figura 2.3 e la seguente *tavola di verità*, dove `0` corrisponde a falso e `1` a vero.

x	**y**	**x&&y**	**x||y**	**!x**
0	0	0	0	1

0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Il connettivo `&&` restituisce vero se e solo se il risultato di entrambe le due operazioni logiche, quella che lo precede e quella che lo segue, risultano vere. L'espressione

```
x==y && a>b
```

è quindi vera se `x` è uguale a `y` e contemporaneamente `a` è maggiore di `b`.

Il connettivo `||` restituisce vero se almeno una delle espressioni logiche che esso concatena risulta vera. Quindi l'espressione

```
b<c || t!=r
```

restituisce falso solamente se `b` non è minore di `c` e contemporaneamente `t` è uguale a `r`.

L'espressione

```
!y
```

restituisce vero se `y` è falso e viceversa. L'operatore `!` è di tipo unario, gli altri sono binari.

L'ordine di priorità complessiva degli operatori logici e relazionali è mostrato in Figura 2.4. Naturalmente, si possono utilizzare le parentesi tonde per alterare la priorità; per esempio, in

```
(nome1!=nome2 || cogno1>cogno2) && presenti < 524
```

viene valutato prima `||` di `&&`.

Il risultato di un'espressione logica può essere assegnato a una variabile, come abbiamo già visto tra gli esempi del primo paragrafo:

```
a = i<100
```

Dalla tabella gerarchica degli operatori deduciamo che l'assegnamento è quello con minor priorità; è per questa ragione che viene valutato innanzitutto `i<100`; se `i` risulta minore di 100 `a` assume valore 1, altrimenti assume valore 0. L'aggiunta di un punto e virgola trasforma l'espressione in una istruzione:

```
a = i<100;
```

Esaminiamo ora un altro esempio:

```
y = a==b;
```

dove `y` assume valore 1 se `a` è uguale a `b`, 0 viceversa. Dunque è lecito anche questo assegnamento, la cui interpretazione lasciamo al lettore:

```
x = (x==y) && (z!=t || m>=n);
```

Dato che le espressioni logiche restituiscono un risultato numerico, non esistono differenze tra le espressioni logiche e quelle aritmetiche; la Figura 2.4 riporta la scala di priorità complessiva degli operatori che abbiamo presentato in questo capitolo ■.

```
!      - (unario)
```

```
*       /       %

+       -

>       >=      <       <=

==      !=

&&

||

?:

=
```

Figura 2.4 Gerarchia degli operatori esaminati in questo capitolo

Un'espressione può contenere una combinazione di operatori aritmetici, logici e relazionali. L'espressione

```
x + sereno + (i<100)
```

restituisce la somma di `x`, di `sereno` e del risultato di `i<100`, che è 1 se `i` è minore di 100, zero altrimenti. Per esempio, se `x` vale 5, `sereno` 6 e `i` 98, l'espressione restituisce 12.

È un'espressione anche la seguente:

```
y = x + sereno + (i<100)
```

dove l'assegnamento a `y` viene effettuato alla fine, perché = ha la priorità più bassa. Essendo l'operatore di assegnamento trattato alla stregua degli altri, sarà lecita anche la seguente espressione:

```
i>n && (x=y)
```

che mette in AND le espressioni `i>n` e `x=y`. La prima è vera se il valore di `i` è maggiore di `n`. La seconda corrisponde all'assegnamento del valore di `x` alla variabile `y`; tale assegnamento viene effettuato all'interno dell'espressione, dopo di che se il valore di `x` è diverso da zero l'espressione risulta vera, altrimenti falsa.

✓ **NOTA**

Queste caratteristiche rendono il C un linguaggio flessibile che consente la scrittura di codice sintetico ed efficiente, ma anche difficilmente interpretabile. Prendiamo ad esempio il seguente frammento di codice:

```
if((x=y)  && i>n)
  printf("vero");
else
  printf("falso");
```

L'espressione presente nell'`if` contiene (nasconde!) un assegnamento. È chiaro che in questo modo risparmiamo una linea di codice ma, ci pare, a scapito della chiarezza. In particolare, si faccia attenzione a non confondere l'operatore relazionale == con quello di assegnamento. Dati i due casi:

```
if(x=y)                    if(x==y)
  printf("vero");            printf("vero");
else                       else
  printf("falso")            printf("falso");
```

in nessuno dei due il compilatore segnalerà un errore, poiché le espressioni presenti sono sintatticamente corrette. Nel caso di sinistra, nell'`if` il valore di `y` viene assegnato a `x` e se risulta essere diverso da zero viene stampato vero, altrimenti falso. Nel caso di destra non avviene nessun assegnamento: se `x` risulta uguale a `y` verrà stampato vero, altrimenti falso. L'effettuare un assegnamento non voluto è uno degli errori più frequenti e quindi vale la pena rimarcare un'ultima volta che l'operatore di confronto di uguaglianza è ==.

Nell'espressione

```
(x=y)  && i>n
```

non è stato necessario racchiudere `i>n` tra parentesi tonde perché l'operatore > ha una priorità maggiore di `&&`. A volte, in special modo nelle espressioni molto lunghe, può risultare difficile comprendere a prima vista, in fase di programmazione, l'ordine di valutazione, per cui niente ci vieta di aggiungere delle parentesi tonde per rafforzare il concetto che vogliamo esprimere:

```
(x=y) && (i>n)
```

Per esempio,

```
((x) && (a>b)  || ((c<=d) && (f!=r))
```

è certamente equivalente a

```
x && a>b || c<=d && f!=r
```

ma la prima forma ci sembra più leggibile e può facilitare la revisione del programma.

2.4.3 Espressioni condizionali

Una *espressione condizionale* si ottiene con l'operatore ternario `?:` che ha la seguente sintassi:

espr1 `?` *espr2* `:` *espr3*

Se la valutazione di *espr1* restituisce vero, il risultato è uguale a *espr2*, altrimenti è uguale a *espr3*. Per esempio,

```
x==y ? a : b
```

significa: "se `x` è uguale a `y`, allora `a`, altrimenti `b`".
Si può utilizzare l'operatore `?:` per assegnare un valore a una variabile, come nel caso che segue:

```
v=x==y ? a*c+5 : b-d;
```

Se `x` è uguale a `y`, a `v` viene assegnato il valore di `a*c+5`, altrimenti gli viene assegnato il valore di `b-d`. Le espressioni *espr1*, *espr2* ed *espr3* vengono valutate prima di `?:` e l'assegnamento viene effettuato dopo, data la posizione dell'operatore condizionale nella tavola gerarchica.
L'espressione condizionale può essere sempre sostituita da un `if` corrispondente: nel caso precedente, per esempio, avremmo potuto scrivere:

```
if(x==y)
  v = a*c+5;
else
  v = b-d;
```

Essendo quella condizionale un'espressione come tutte le altre, può essere inserita in qualsiasi posizione sia appunto lecito scrivere un'espressione. Potremmo cioè scrivere:

```
x = a*x+(x==z ? b : c)+d;
```

corrispondente a

```
if(x==z)
  x = a*x+b+d;
else
  x = a*x+c+d;
```

L'istruzione

```
printf("%d maggiore o uguale a %d", (a>b?a:b), (a<=b?a:b));
```

restituisce:

- se a ha valore 5 e b ha valore 3:
  `5 maggiore o uguale a 3`
- se a è uguale a 100 e b è uguale a 431:
  `431 maggiore o uguale a 100`
- se a è uguale a 20 e b è uguale a 20:
  `20 maggiore o uguale a 20`

L'operatore `?:` può essere inserito in qualsiasi espressione, dunque anche all'interno della condizione che controlla l'`if`:

```
if((a>=b ? a : b) >= c )
  printf("nero");
else
  printf("bianco");
```

In generale, quando più operatori con la stessa priorità devono essere valutati in un'espressione, lo standard C non garantisce l'ordine di valutazione. Nell'espressione

```
x > y <= z
```

non possiamo sapere se verrà valutata prima la condizione `x > y` e successivamente se il risultato (zero o uno) è minore o uguale a `z`, o viceversa. In ogni caso è buona norma non scrivere codice dipendente dall'ordine di valutazione per non rischiare di produrre programmi non portabili tra versioni C differenti.

Esiste comunque un'eccezione. Per gli operatori `&&`, `||`, e `?:` il linguaggio garantisce la valutazione delle espressioni da sinistra verso destra. Per esempio, in

```
x==y && a>b
```

l'espressione `a>b` sarà valutata soltanto se il risultato dell'espressione `x==y` è vero. Analogamente in

```
b<c || t!=r
```

l'espressione `t!=r` sarà valutata soltanto se il risultato dell'espressione `b<c` è falso.

Nell'espressione

```
x>y ?  a>b ?  a : b : y
```

prima viene valutata `a>b`, quindi restituito `a` oppure `b`; successivamente è valutata `x>y`, dato che `?:` è associativo da destra verso sinistra.