Espressioni

Espressioni

Una espressione E del linguaggio C può essere definita formalmente come segue (definizione induttiva):

- E è una *espressione semplice*.
- Sia Op_n un *operatore* del linguaggio di arità n ($n = 1, 2, 3$) e siano $E_1, \ldots, E_n$ delle espressioni. Allora

  $$Op_n(E_1, \ldots, E_n)$$

  è una espressione (*espressione composta*).

A seconda dell'operatore, si usa la notazione *prefissa*, *infissa* o *postfissa*.

Esiste un solo operatore Op_3 ternario, che è l'operatore condizionale:

```
E1 ? E2 : E3
```


Valutazione di un'espressione

Ogni espressione E ha un tipo T e un valore (di tipo T).

- Il tipo di un'espressione può essere determinato dall'analisi statica del programma in quanto dipende solamente dalle dichiarazioni presenti nel programma (type-check statico).
- Valutare E significa calcolare il valore di E.
  - Il valore di un'espressione dipende dallo stato del sistema (valori presenti nella memoria utilizzata dal programma) nel momento in cui avviene la valutazione
  - La valutazione di una espressione può produrre effetti collaterali (side-effect), ossia un cambiamento dello stato del sistema.

Nel seguito, Valore(E) indica il valore dell'espressione E.

Espressioni semplici

- Una costante è una espressione semplice, il cui tipo e valore sono quelli denotati dalla costante.

  Esempio

  La costante `12L` *ha tipo* `long int` *e* *Valore*(`12L`) $= 12$.

  La costante `12.0` *ha tipo* `double` *e* *Valore*(`12.0`) $= 12$.

  Il valore di un'espressione costante non dipende dallo stato del sistema.
- Una variabile è una espressione semplice, il cui tipo è il tipo con cui la variabile è stata dichiarata e il valore dipende dallo stato del sistema nel momento in cui avviene la valutazione.

  Esempio

  Dopo le istruzioni

  ```
  int x = 10;
  x = x-4;
  ```

  l'espressione `x` ha tipo `int` e valore 6.

Tabella degli operatori

Operatori	Associatività
`()` (*funzione*) `[]` (*array*) `.` (*campo di una struttura*) `->` (*dereferenziazione di struttura*) `++` (*postfisso*) `--` (*postfisso*)	⇒
`++` (*prefisso*) `--` (*prefisso*) `+` (*unario*) `-` (*unario*) `sizeof` `()` (*cast*) `!` (*not*) `&` (*indirizzo*) `*` (*derferenziazione*)	⇐
`*` (*prodotto*) `/` (*quoziente*) `%` (*modulo*)	⇒
`+` (*somma*) `-` (*differenza*)	⇒
`<` `<=` `>` `>=` (*confronto*)	⇒
`==` `!=` (*confronto*)	⇒
`&&` (*and*)	⇒
`||` (*or*)	⇒
`?:` (*condizionale*)	⇐
`=` `+=` `-=` `*=` `/=` `%=` (*assegnamento*)	⇐
`,` (*virgola*)	⇒

Priorità di un operatore

- Determina l'ordine con cui gli operatori vanno applicati; per primi si applicano quelli con priorità più alta per ultimi quelli con priorità più bassa.
- Nella tabella, gli operatori a priorità più alta sono quelli nella prima riga, quelli a priorità più bassa sono quelli nell'ultima riga.
  Gli operatori nella stessa riga della tabella hanno la stessa priorità.

Esempio

Poiché < ha priorità maggiore di =, l'espressione

```
x = y < z
```

va interpretata come

```
x = (y < z)
```


Associatività di un operatore

- Determina in che ordine vanno applicati gli operatori aventi la stessa priorità.
- L'associatività può essere "da sinistra a destra" oppure "da destra a sinistra" (è indicata dalla freccia nella seconda colonna).

Esempio

- L'operatore < ha associatività ⇒.
  L'espressione
  ```
  x < y < z
  ```
  equivale a
  ```
  (x < y) < z
  ```

- L'operatore di assegnamento = ha associatività ⇐.
  L'espressione
  ```
  x = y = z
  ```
  equivale a
  ```
  x = (y = z)
  ```


Uso delle parentesi

Le espressioni fra parentesi tonde vengono valutate per prima. Quindi, le parentesi vengono utilizzate per *modificare* l'ordine di valutazione degli operatori (oppure, per evidenziare l'ordine con cui gli operatori vengono valutati).

Esempio

- Nell'espressione
  ```
  (x || y) && z
  ```
  le parentesi sono necessarie, altrimenti l'espressione verrebbe interpretata come
  ```
  x || (y && z)
  ```
  avendo && priorità maggiore di ||.

- Nell'espressione
  ```
  (a && b ) || (c && d)
  ```
  le parentesi possono ossere omesse, ma l'uso delle parentesi rende più leggibile l'espressione.

Operatori per il confronto di espressioni

- Per confrontare il valore di due espressioni si usano gli operatori binari di confronto `<` (*minore*), `<=` (*minore o uguale*), `>` (*maggiore*), `>=` (*maggiore o uguale*), `==` (*uguale*), `!=` (*diverso*).
- In C non esiste il tipo booelan, quindi E_1 *Op* E_2 è vista (in modo "poco astratto") come una *espressione intera*.

Più precisamente:

- E_1 *Op* E_2 ha tipo `int`.
- Il valore dell'espressione è definito come segue:

$$\text{Valore}(E_1\ Op\ E_2) = \begin{cases} 1 & \text{se Valore}(E_1)\ Op\ \text{Valore}(E_2) \text{ è vero;} \\ 0 & \text{altrimenti.} \end{cases}$$

Esempi

Espressione	Valore
`5 < 3`	0
`7.23 > (3+1)`	1
`'a' < 'f'`	1
`'a' + 2 > 'b'`	1
`98.1 - 1.1 == 'a'`	1
`(5.5 > -2) + 3`	4 (tipo `int`)
`(5.5 > -2) + 3.0`	4.0 (tipo `double`)
`(10 <= 11)== 1`	1
`(10 <= 11)== 0`	0

Attenzione

Si supponga di dover tradurre in C l'istruzione di alto livello

if $10 < x < 20$
then (A)
else (B)

Si vuole cioè eseguire (A) se il valore di x è strettamente compreso fra i valori 10 e 20, (B) altrimenti.

Soluzione errata

```
if(10 < x < 20)
  (A)
else
  (B)
```


- Avendo < associatività da sinistra a destra,
  `10 < x < 20`
  equivale a
  `(10 < x) < 20`
- Il valore dell'espressione
  `10 < x`
  è 1 (valore di 10 <100).
- Il valore dell'espressione
  `(10 < x) < 20`
  è 1 (valore di 1<20).

Viene quindi eseguita istruzione (A).

Soluzione corretta

Occorre fare l'AND logico delle condizioni $10 < x$ e $x < 20$:

```
if( (10 < x) && (x< 20) )
  (A)
else
  (B)
```

Esercizio

Dire qual è il comportamento di

```
if(x == y == z)
  (A)
else
  (B)
```


L'operatore di assegnamento =

Se x è una variabile di tipo T, l'espressione

$$x = Espr$$

- Ha tipo T (tipo di x).
- La valutazione di $x = Espr$ avviene come segue.
  - Viene valutato il valore v di $Espr$.
  - Viene fatto un cast al tipo T di v.
  - Viene assegnato a x il valore v (*side-effect* della valutazione).

  Il valore di $x = Espr$ è v (visto come valore di tipo T).

- In C l'assegnamento è visto come operatore, a differenza di quanto avviene usualmente nei linguaggi di programmazione in cui l'assegnamento è un'istruzione.
  Questo permette di eseguire assegnamenti all'interno di altre espressioni, ad esempio:
  ```
  (c=getchar()) != EOF
  ```
- In generale (vedere i dettagli sui manuali):
  - a *sinistra* di un operatore di assegnamento può esserci un qualunque lvalue (*left-value*), ossia un'espressione E_{left} che denota una locazione di memoria.
    In questo caso, il tipo T dell'espressione di assegnamento è il tipo di E_{left}.
  - a *destra* di un operatore di assegnamento può esserci una qualunque espressione, detta rvalue (*right-value*).
- L'assegnamento è un esempio di operatore con *side-effect*.
  La valutazione di espressioni con all'interno degli assegnamenti in genere cambia lo stato del sistema.

Esempi

```
i int; d double;
```

- La valutazione dell'espressione
  ```
  i = 7
  ```
  ha come effetto collaterale quello di assegnare a i il valore 7; l'espressione ha tipo int e valore 7.
- La valutazione dell'espressione
  ```
  d = 7
  ```
  provoca un cast implicito di 7 a double (tipo di d).

  Le seguenti espressioni sono equivalenti
  ```
  d = 7        d = (double)7        d =  7.0
  ```
  Tutte hanno tipo double e valore 7 e, come side-effect della valutazione, a d viene assegnato il valore 7.

Per verificare ciò, scrivere un programma contenente le istruzioni

```
printf("%f", d=7); // stampa il valore dell'espressione d=7 con il formato %f

printf("%d", d=7); // stampa il valore dell'espressione d=7 con il formato %f

printf("%d", (int)(d=7));  //  conversione esplicita a int del valore di d=7
```

Compilarlo con opzione -Wall ed eseguirlo.

Esercizio

Eseguire passo-passo il seguente programma.

```
void f(int x, int y){
 x = x+y;
 y = x-y;
}

int main(){
 int x, y;
 f(y=20, x=10);  // attenzione al passaggio dei parametri!
 return 0;
}
```

Ricordarsi che il passaggio dei parametri avviene nell'ambiente del chiamante (e non del chiamato). Quindi, la `x` e la `y` nominati in `main()` si riferiscono alle variabili `x` e `y` locali `main()` (e non alle variabili locali a `f()`).

Altri operatori di assegnamento

- Sono gli operatori della forma $\odot =$, dove $\odot$ è uno degli operatori

  `+` `-` `*` `/` `%` ...

  Ad esempio:

  `+=` `-=` `*=` `/=` `%=`

- L'espressione

  $$Var \quad \odot = \quad Espr$$

  è equivalente a

  $$Var \quad = \quad Var \odot (Espr)$$

Esempio

`x += 3`	equivale a	`x = x + 3`
`y *= z + 5`	equivale a	`y = y * (z + 5)`
`a %= a * (x +y)`	equivale a	`a = a % (a * (x +y))`

Gli operatori di assegnamento hanno priorità bassa (ultima riga della tabella) e associatività da destra a sinistra.

Esempio

- L'espressione

  `x = y = 3`

  viene interpretata come

  `x = (y = 3)`

  Per valutarla, occorre prima valutare `y = 3`, che provoca l'assegnamento del valore 3 a `y`. Successivamente, a `x` viene assegnato il valore dell'espressione `y = 3`, che è 3.

- Supponendo che `x` valga 3, dopo la valutazione dell'espressione

  `x *= (z = 4) + 6`

  equivalente a

  `x = x * ((z = 4) + 6)`

  `z` vale 4 e `x` vale 30.

  Per stampare il valore dell'espressione:

```
int x=3, z;
printf("%d", x *= (z = 4) + 6);
```


Divisione intera e reale

Esempio 1

Si vuole calcolare la media dei voti di uno studente.

```
int somma,   // somma dei voti
int  n;  // numero esami sostenuti
double media;
```

Soluzione errata

```
media = somma / n;
```

Infatti, se `somma` vale 289 e `n` vale 10:

- il valore di

  `somma / n`

  è 28 (risultato della divisione intera).

- A `media` è assegnato il valore 28 convertito a `double`.

L'istruzione è equivalente a

```
media = (double) (somma / n);
```

Il cast a `double` viene fatto *dopo* aver fatto la divisione intera.

Soluzione corretta

Bisogna fare in modo che eseguita la divisione fra reali (e non la divisione fra interi).

L'istruzione va scritta in uno dei seguenti modi equivalenti:

```
media = (double) somma / n;

media = somma / (double) n;

media = (double) somma / (double) n;
```

In questo modo alla variabile `media` viene assegnato il valore corretto 28.9.

Esempio 2

Supponiamo di voler estrarre la prima cifra c_1 (cifra più significativa) di della rappresentazione in base 10 di un intero positivo n.

Allora:

- Si calcola il numero di cifre k di n.
- Si divide n per 10^{k-1} (divisione intera!).

Il numero di cifre k di un intero $n > 0$ è dato dalla formula

$$k = \lfloor \log_{10}(n) \rfloor + 1$$

dove $\lfloor r \rfloor$ è la parte intera di r e può essere calcolato dalla funzione `floor()` di `math.h`.

```
/* Calcola il numero di cifre di un intero n>0 (notazione decimale) */

int numCifre(int n){
  return floor(log10(n)) + 1;  // floor() calcola la parte intera
                               // log10() calcola il logaritmo in base 10
}
```


Soluzione errata

```
int c1, k;
k = numCifre(n);
c1 = n / pow(10, k-1);  // pow(a,b) calcola a^b
```

Poiché il valore restituito da `pow()` ha tipo `double`, il valore di `n` viene convertito a `double` e viene effettuata la divisione fra reali.

Soluzione corretta

Occorre forzare la divisione intera, facendo un cast del risultato di `pow()` (che è un intero) al tipo `int` (o a altro tipo intero).

```
int c1, k;
k = numCifre(n);
c1 = n / (int) pow(10, k-1);
```

Esercizio

Scrivere il codice della funzione

```
int  radice(int n, double x, double* result)
```

che, se $\sqrt[n]{x}$ è definita, pone il risultato dell'operazione in `*result` (passaggio per indirizzo) e restituisce 1, altrimenti restituisce 0 (si ricordi che $\sqrt[n]{x} = x^{1/n}$).

Gli operatori di incremento e decremento

- Sono definiti l'operatore di incremento `++` *prefisso*, l'operatore di incremento `++` *postfisso*, l'operatore di decremento `--` *prefisso* e l'operatore di decremento `--` *postfisso*.
- Vedere sui manuali a quali espressioni possono essere applicati.
- Hanno priorità molto alta.

Espressione	Valore	Effetto collaterale prodotto dalla valutazione
`++x`	`x+1`	`x` viene incrementato di uno
`x++`	`x`	`x` viene incrementato di uno
`--x`	`x-1`	`x` viene decrementato di uno
`x--`	`x`	`x` viene decrementato di uno

Esempi

```
int a=0, b=0, c=0;

a = c++; // (1)
b = ++a;  // (2)
c = b-- + --a;   // (3)
// equivale a:  c = ((b--) + (--a));
```

- Quando viene eseguita l'istruzione (1), viene valutata l'espressione `a = c++`. Al termine della valutazione ad `a` viene assegnato 0 e `c` vale 1 (`c` viene incrementato *dopo* aver utilizzato il suo valore).
- Con l'istruzione (2) viene valutata l'espressione `b = ++a`. A `b` viene assegnato 1 e `a` vale 1 (`a` viene incrementato *prima* che venga utilizzato il suo valore).
- Dopo (3), `c` vale 1 (valore di $1 + 0$), `b` vale 0 e `a` vale 0.

Le seguenti espressioni non sono corrette.

```
int x;
(x++) = 20;   // non si puo' incrementare locazione di memoria a sin. di =
x = 20--;     // non si puo' decrementare una costante
++(x++);      // ++ esterno e' applicato a una costante
```


Istruzioni ambigue

- L'ANSI C non specifica quando, nella valutazione di una espressione, debbano di fatto avvenire i side-effect provocati dagli operatori di incremento e decremento.
- L'ANSI C non stabilisce in che ordine debbano essere valutati gli argomenti di una funzione.

Questo comporta che alcune espressioni siano ambigue in quanto il loro valore dipende dal sistema su cui sono valutate.

Esempio 1

```
int a = 10;
b =  a++ + a;
```

Il valore dell'espressione `a++` è 11 (valore dell'espressione `a+1`).

Il valore dell'espressione `a` a destra di `+` dipende da quando l'incremento di `a` (side-effect della valutazione di `a++`) è effettuato:

- se l'incremento è fatto subito dopo la valutazione di `a++`, `a` vale 11, quindi $b = 10 + 11$.
- se l'incremento è fatto dopo il calcolo della somma `a++ + +a`, il valore di `a` è 10, quindi $b = 10 + 10$.

Esempio 2

```
int n = 5;
printf("..%d..%d.. ", n++, n++);
// n vale 7
```

- Se la valutazione degli argomenti avviene da sinistra a destra, la chiamata `printf()` equivale a
  ```
  printf("..%d..%d.. ", 5, 6);
  ```
- Se la valutazione degli argomenti avviene da destra a sinistra equivale a
  ```
  printf("..%d..%d.. ", 6, 5);
  ```

Occorre evitare situazioni di questo tipo scrivendo in modo non ambiguo quello che si vuole ottenere.
Ad esempio, se si vuole la prima soluzione, occorre fare:

```
int n = 5;
printf("..%d..%d.. ", n, n+1);
n = n+2;
```


L'operatore condizionale

- L'operatore condizionale '`?:`' è l'unico operatore *ternario*.
- Ha priorità più alta degli operatori di assegnamento e più bassa degli operatori logici e associatività $\Leftarrow$.

La valutazione dell'espressione

$$Espr_1 \quad ? \quad Espr_2 \quad : \quad Espr_3$$

avviene nel seguente modo:

- Viene valutata l'espressione $Espr_1$.
- Se Valore($Espr_1$) $\neq 0$:
  - Viene valutata l'espressione $Espr_2$.
  - Il tipo e il valore dell'intera espressione coincide con il tipo e il valore di $Espr_2$.

  Se Valore($Espr_1$) $= 0$:
  - Viene valutata l'espressione $Espr_3$.
  - Il tipo e il valore dell'intera espressione coincide con il tipo e il valore di $Espr_3$.

Esempio

La funzione

```
int f(int x, int y){
  return  x > y  ?  x  :  y;
 }
```

restituisce il massimo fra x e y.

Equivale a:

```
 int f(int x, int y){
  if(x>y)
    return x;
  else
    return y;
 }
```

Esercizio

Dire se la funzione

```
int g(int x, int y){
  return  x >  (y  ?  x  :  y);
 }
```

è corretta e, in caso di risposta affermativa, cosa calcola.

Operatori logici

- Anche se in C non è definito il tipo boolean per rappresentare i valori di verità *vero* e *falso*, esistono degli operatori che si comportano come i connettivi logici booleni: l'operatore unario `!` (*not*) e gli operatori binari `&&` (*and*) e `||` (*or*).
- Tutti e tre gli operatori si applicano a espressioni arbirarie e producono espressioni di tipo `int` e valore 0 oppure 1.
- L'operatore `!` ha priorità molto alta.
  Gli operatori `&&` e `||` hanno priorità bassa e `&&` ha priorità maggiore di `||`.

Definizione dei valori

$$\text{Valore}(!\ \mathit{Espr}) \quad = \quad \begin{cases} 1 & \text{se Valore}(\mathit{Espr}) = 0 \\ 0 & \text{altrimenti} \end{cases}$$

$$\text{Valore}(E_1\ \texttt{\&\&}\ E_2) \quad = \quad \begin{cases} 1 & \text{se Valore}(E_1) \neq 0 \text{ e Valore}(E_2) \neq 0 \\ 0 & \text{altrimenti} \end{cases}$$

$$\text{Valore}(E_1\ \texttt{||}\ E_2) \quad = \quad \begin{cases} 1 & \text{se Valore}(E_1) \neq 0 \text{ oppure Valore}(E_2) \neq 0 \\ 0 & \text{altrimenti} \end{cases}$$

Esempio

Il seguente programma stampa una tabella degli operatori logici.

```
void intestazione(){// stampa prime due righe dell'output
printf("...");
}

int main(){
   char b1, b2;           // variabili  "booleane"
   intestazione();
   for (b1 = 0; b1 <= 1; b1++)
      for (b2 = 0; b2 <= 1; b2++)
          printf("%5d %5d %5d %5d %5d %8d\n",
                 b1, b2, !b1, !b2,  b1&&b2, b1||b2);
   putchar('\n');
   return 0;
}
```

L'output è:

```
   b1    b2    !b1   !b2   b1&&b2   b1||b2

    0     0     1     1     0         0
    0     1     1     0     0         1
    1     0     0     1     0         1
    1     1     0     0     1         1
```

Esempi

```
char c=98; int i=100; double d=7.5;
```

Espressione	Tipo	Valore
`c && i`	`int`	1
`c && i + d * 10`	`double`	76.0
`c && (i - 100)`	`int`	0
`'c' > c > 'a'`	`int`	0
`'c' > c && c > 'a'`	`int`	1
`d || d-d`	`int`	1
`(d || d)-d`	`double`	-6.5
`!d`	`int`	0
`!!d`	`int`	1
`!( c < 'c')`	`int`	0
`!c < 'c'`	`int`	1
`!!c == 1`	`int`	1
`!'b' || 'b'-c`	`int`	0
`!'d' && d || 'd' && d`	`int`	1
`!'d' && (d || 'd') && d`	`int`	0

Short-Circuit Evaluation

Se il valore dell' espressione

$$Espr_1 \diamond Espr_2 \qquad \diamond \in \{\&\&,\ ||\}$$

è deducibile dal valore di $Espr_1$, l'espressione $Espr_2$ non viene valutata. Questa modalità di valutazione è chiamata Short-Circuit Evaluation (valutazione cortocircuitata) e funziona come in Java.

Formalmente:

- L'espressione
  $$Espr_1 \quad \&\& \quad Espr_2$$
  equivale all'espressione ternaria
  $$(Espr_1 == 0) \quad ? \quad 0 \quad : \quad (Espr_2 \; != 0)$$
- L'espressione
  $$Espr_1 \quad || \quad Espr_2$$
  equivale all'espressione ternaria
  $$(Espr_1 \; != 0) \quad ? \quad 1 \quad : \quad (Espr_2 \; != 0)$$

Non commutatività operatori logici

Come conseguenza della valutazione cortocircuitata, in presenza di side-effect i due connettivi logici binari non sono commutativi (a differenza dei corrispondenti connettivi logici) e occorre prestare attenzione all'ordine con cui vengono scritti gli operandi.

Esempio

```
if( x!=0 && z = y/x)
  ...
```

L'assegnamento

```
z = y/x
```

viene effettuato solo se l'espressione `z = y/x` è valutata, quindi solo se il valore di `x` è diverso da 0.
Invece, in

```
if ( z = y/x  &&  x!=0 )
  ...
```

l'espressione `z = y/x` viene *sempre* valutata.

Se `x` vale 0, la divisione `y/x` provoca errore in esecuzione

Altri tipi di espressione

Nel corso si sono introdotte anche le seguenti espressioni:

- Espressioni ottenute mediante gli operatori di indirizzamento '`&`' e dereferenziazione '`*`' e '`[]`'.

  Si ricordi che, data una variabile `p` di tipo `int*`

  ```
  p[4]
  ```

  è una espressione il cui operatore principale '`[]`' è applicato alle espressioni `p` e 4, ed è equivalente a

  ```
  *(p+4)
  ```

- Chiamate di funzioni, che in notazione BNF (vedere i manuali) hanno la forma

  ```
  function_name(argument_list)
  ```

  In questo caso l'operatore principale è la coppia di parentesi tonde `()`, che ha come sottoespressioni il nome della funzione e la lista degli argomenti con cui è chiamata.

Istruzioni

Istruzione semplice

Un'istruzione semplice può essere:

- l'istruzione vuota, denotata da ';'.
- un'espressione seguita da ';'.

Sintassi:
Un'istruzione semplice è definita da

$$\text{<istruzione\_semplice>} ::= \{ \text{<espr>} \}_{opt} \ ;$$

Le parentesi '$\{\ldots\}_{opt}$' significano che il termine <espr> è opzionale (nel caso manchi, si ha l'istruzione vuota).

Semantica:
Viene valutata l'espressione <espr>.

Esempio

```
int a=0, b=1, c=2;

a = b + (c=4);  // (1)
a < b;   // (2)
printf("%d",a); // (3)
;   //  (4)  (istruzione vuota)
```

Quando il programma viene eseguito, vengono valutate in successione le espressioni (1),(2),(3) e (4) producendo i seguento effetti:

- *Istruzione (1)*: al termine della valutazione c vale 4, b vale 1 e a vale 5.
- *Istruzione (2)*: non produce alcun effetto.
- *Istruzione (3)*: viene stampato 5; il valore 1 restituito da `printf()` (numero dei caratteri stampati) non viene utilizzato.
- *Istruzione (4)*: non fa nulla.

Blocco

- Le graffe vengono usate per racchiudere istruzioni che formano un'istruzione composta o blocco.
- Un blocco è un'istruzione, quindi in un programma dove è sintatticamente corretto scrivere un'istruzione si può scrivere un blocco.

$$\text{<blocco>} ::= \{ \{ \text{<dichiarazioni>} \}_{0+} \{ \text{<istruzione>} \}_{0+} \}$$

- Questo significa che in un blocco ci possono essere zero o più dichiarazioni di variabili seguite da zero o più istruzioni.
- Una variabile è visibile solo nel blocco in cui è definita e negli eventuali blocchi interni. Il nome interno nasconde il nome esterno.

Esempio

```
{
 ...
  {// blocco A
    int a,b;
    a = 10;
    b = 20;
    { // blocco B
      int b;   // nasconde la b del blocco A
      int c;
      b = 2; // modifica il valore di b del blocco B
      a = 7; // modifica il valore di a del blocco A
    }  // end  blocco B
   b = 5; // le variabili del blocco B non sono piu' visibili
  } // end blocco A
 ...
}
```


Strutture di controllo

Nelle strutture di controllo, la condizione di terminazione è espressa da una espressione di qualunque tipo (non essendoci in C le espressioni booleane).

La struttura di controllo "if-then-else"

La struttura "if-then-else" corrisponde all'istruzione composta del C <if_else_ist>.

Sintassi:

<if_else_ist> ::= `if` (<espr>) <istruzione-1> `else` <istruzione-2>

dove il termine <istruzione-...> denota una qualunque istruzione.

Semantica:

- Viene valutata <espr>.
  - Se Valore(<espr>) $\neq 0$
    viene eseguita <istruzione-1>.
  - Se Valore(<espr>) $= 0$
    viene eseguita <istruzione-2>.

Esempio

```
/* (1)  */              /*  (2)  */      /*  (3)  */
if(a)                  if(a == 0)       if(a=0)
 printf("V");           printf("V");     printf("V");
else                   else             else
 printf("F");           printf("F");     printf("F");
```

Le tre istruzioni (riportate su tre colonne) hanno comportamenti diversi (verificarlo). Infatti:

- L'istruzione (1) stampa `V` se il valore di a è diverso da 0, `F` se a vale 0.
- L'istruzione (2) stampa `V` se a vale 0, `F` se a è diverso da 0 (infatti, se a vale 0 l'espressione a==0 vale 1, altrimenti a==0 vale 0).
- L'istruzione (3) stampa `F`. Infatti, come effetto della valutazione dell'espressione a=0, ad a viene assegnato il valore 0 e il valore dell'espressione in `if` è 0.

Attenzione

Non confondere l'operatore di assegnamento = con l'operatore di confronto ==.

La struttura di controllo "if-then"

È analoga alla struttura "if-then-else", in cui però non c'è il ramo "else"; in C corrisponde all'istruzione composta <if_ist>.

Sintassi:

<if_ist> ::= `if` (<espr>) <istruzione>

Semantica:

- Viene valutata <espr>.
  - Se Valore(<espr>) $\neq 0$:
    viene eseguita <istruzione>.
  - Se Valore(<espr>) $= 0$:
    l'esecuzione riprende dall'istruzione successiva a <if_ist>.

Esempio

```
if (l>=0)
   area = l*l;
printf("Area: %d", area);
...
```

Solo la prima istruzione fa parte del corpo di `if`.

Se si vuole che l'istruzione `printf(...)` venga eseguita solamente se $l \geq 0$, occorre introdurre un blocco.

```
if (l>=0) {
   area = l*l;
   printf("Area: %d", area);
  }
...
```


La struttura di controllo "while"

Corrisponde all'istruzione composta <while_ist>.

Sintassi:

<while_ist> ::= `while` (<espr>) <istruzione>

Semantica:

(1). Viene valutata <espr>.
- Se Valore(<espr>) $\neq 0$:
  viene eseguita <istruzione> e il controllo torna al punto (1).
- Se Valore(<espr>) $= 0$:
  il controllo passa all'istruzione che segue <while_ist>.

È inoltre definita la struttura di controllo `do..while`, in cui la condizione associata al ciclo è valutata al termine di ogni iterazione.

Esempio 1

```
int n = 1;
   while(n++ < 100)
       printf("%d\n", n); // (1)
   printf("Valore finale: %d\n", n); // (2)
```

L'effetto del codice è quello di stampare le linee

```
1
2
...
100
Valore finale: 101
```

- Nel ciclo `while` c'è solo l'istruzione (1), al termine del ciclo viene eseguita (2) (se a ogni iterazione deve essere eseguita più di una istruzione occorre usare le parentesi graffe).
- L'espressione `n++ < 100` vale 1 *se e solo se* il valore di `n` è minore di 100; a ogni valutazione di tale espressione, `n` viene incrementato di 1.
- Quando `n` vale 100, `n++ < 100` vale 0 (come effetto della valutazione la variabile `n` assume il valore 101), quindi il ciclo termine e viene eseguita l'istruzione (2).

Esempio 2

```
int n= 1;
while(n++<100);
    printf("%d",n);
```

In questo caso l'istruzione dentro il ciclo non è `printf(...)`, ma l'istruzione vuota ";".

È come se fosse scritto

```
while(n++<100)
 {};
 printf("%d",n);
```

- A ogni iterazione viene valutata l'espressione `n++ < 100` (con conseguente incremento di `n`).
- Quando `n` raggiunge il valore 100, l'espressione `n++ < 100` vale 0 e il valore di `n` è incrementato di 1.
- Il controllo riprende dall'istruzione dopo l'istruzione `while` e viene stampato 101 (valore di `n`).

Esempio 3

Vengono spesso usati cicli `while` del tipo:

```
while( (c=getchar()) != EOF){
  (A)
}
(B)
```

La valutazione dell'espressione associata a `while` avviene in questo modo:

- Viene valutata l'espressione `c=getchar()`.
  La valutazione ha come effetto la chiamata a `getchar()` e l'assegnamento a c del valore *n* restituito da `getchar()` (corrispondente al carattere letto da standard input).
  Il valore dell'espressione è *n*.
- Se *n* è diverso da `EOF`, l'espressione in `while` ha valore 1; viene eseguito (A) e successivamente si torna al punto (1).
  Se *n* è uguale a `EOF`, il controllo passa a (B).

Le parentesi sono necessarie. Infatti

```
while( c = getchar() != EOF)
```

equivale a

```
while( c = (getchar() != EOF) )
```

avendo `!=` priorità maggiore di `=`.

In questo caso, a c viene assegnato:

- 1 se il valore di `getchar()` è diverso da `EOF`;
- 0 se il valore di `getchar()` è `EOF`.

Il ciclo termina quando a c viene assegnato il valore 0.

Esempio

```
while((c = getchar()) != EOF)
    putchar(c);
```

equivale a

```
c = getchar();
while( c != EOF){
    putchar(c);
    c = getchar();
 }
```


Esempio 4

```
int n;
fatt = 1;
while( n>=2 && (fatt *= n--) );
```

Il ciclo `while` equivale a:

```
while(n>=2){
  fatt = fatt * n;
  n = n-1;
 }
```

Al termine del ciclo il valore di `fatt` è il fattoriale di `n`.

Esercizio

Cosa fa il seguente programma?

```
#include <stdio.h>

int main(){
    short int s=0;
    while(printf("s= %d\n", s += 10000));
}
```

Verificare la risposta eseguendolo.