Programmazione 1

Lezione 3

Vincenzo Marra

vincenzo.marra@unimi.it

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

15 marzo 2017

Le espressioni

- In C, le espressioni si ottengono combinando variabili o costanti tramite gli operatori messi a disposizione dal linguaggio.

Le espressioni

- In C, le espressioni si ottengono combinando variabili o costanti tramite gli operatori messi a disposizione dal linguaggio.
- Poiché variabili e costanti hanno un tipo, anche le espressioni hanno un tipo. Le espressioni hanno anche un valore, ma solo a seguito di una loro valutazione.

Le espressioni

- In C, le espressioni si ottengono combinando variabili o costanti tramite gli operatori messi a disposizione dal linguaggio.
- Poiché variabili e costanti hanno un tipo, anche le espressioni hanno un tipo. Le espressioni hanno anche un valore, ma solo a seguito di una loro valutazione.
- In C è permesso mescolare tipi diversi nella stessa espressione; ne segue che la determinazione del tipo di un'espressione può risultare complessa. Ne parleremo più approfonditamente in seguito. Per ora ci basterà imparare a usare le espressioni in modo elementare, in combinazione con le variabili.

Le espressioni

- In C, le espressioni si ottengono combinando variabili o costanti tramite gli operatori messi a disposizione dal linguaggio.
- Poiché variabili e costanti hanno un tipo, anche le espressioni hanno un tipo. Le espressioni hanno anche un valore, ma solo a seguito di una loro valutazione.
- In C è permesso mescolare tipi diversi nella stessa espressione; ne segue che la determinazione del tipo di un'espressione può risultare complessa. Ne parleremo più approfonditamente in seguito. Per ora ci basterà imparare a usare le espressioni in modo elementare, in combinazione con le variabili.
- Sintatticamente, un'espressione a sé stante non è un'istruzione.

Le espressioni

- In C, le espressioni si ottengono combinando variabili o costanti tramite gli operatori messi a disposizione dal linguaggio.
- Poiché variabili e costanti hanno un tipo, anche le espressioni hanno un tipo. Le espressioni hanno anche un valore, ma solo a seguito di una loro valutazione.
- In C è permesso mescolare tipi diversi nella stessa espressione; ne segue che la determinazione del tipo di un'espressione può risultare complessa. Ne parleremo più approfonditamente in seguito. Per ora ci basterà imparare a usare le espressioni in modo elementare, in combinazione con le variabili.
- Sintatticamente, un'espressione a sé stante non è un'istruzione.
- Tuttavia:

  Un'espressione seguita da un punto e virgola è un'istruzione.

Operatori aritmetici

Operatore	No. Argomenti	Descrizione
+	2	Somma
-	2	Sottrazione
*	2	Prodotto
/	2	Divisione
%	2	Modulo (resto)

Operatori aritmetici

Operatore	No. Argomenti	Descrizione
+	2	Somma
-	2	Sottrazione
*	2	Prodotto
/	2	Divisione
%	2	Modulo (resto)

Nota. Questi operatori si applicano sia a valori integrali che a valori reali, con l'eccezione di %, che può solo essere applicato a tipi integrali. Come abbiamo visto, per esempio, / denota la divisione intera, se gli argomenti sono interi. Se invece uno degli argomenti è reali (`float` o `double`), esso denota la divisione reale. Così, `1/2` vale `0`, ma `1.0/2.0` vale `0.5`, come pure `1.0/2`.

Il Vero e il Falso in C

- In molti linguaggi di programmazione esiste il tipo booleano,[1] i cui dati possono assumere solo due valori distinti convenzionalmente detti `true` e `false`, ossia `vero` e `falso`.

[1] Aggettivo derivato dal nome del matematico e logico inglese George Boole, 1815–1864.

Il Vero e il Falso in C

- In molti linguaggi di programmazione esiste il tipo booleano,[1] i cui dati possono assumere solo due valori distinti convenzionalmente detti `true` e `false`, ossia `vero` e `falso`.
- È importante tenere presente che, tradizionalmente,[2] non esiste un tipo booleano in C.

[1] Aggettivo derivato dal nome del matematico e logico inglese George Boole, 1815–1864.

[2] In verità, lo standard C99 ha introdotto un tipo booleano, ma non ne parleremo.

Il Vero e il Falso in C

- In molti linguaggi di programmazione esiste il tipo booleano,[1] i cui dati possono assumere solo due valori distinti convenzionalmente detti `true` e `false`, ossia `vero` e `falso`.
- È importante tenere presente che, tradizionalmente,[2] non esiste un tipo booleano in C.
- Quando si dice che la valutazione di una certa espressione in C risulta falsa, si intende per convenzione che il risultato è il numero (intero o in virgola mobile) zero.

[1] Aggettivo derivato dal nome del matematico e logico inglese George Boole, 1815–1864.

[2] In verità, lo standard C99 ha introdotto un tipo booleano, ma non ne parleremo.

Il Vero e il Falso in C

- In molti linguaggi di programmazione esiste il tipo booleano,[1] i cui dati possono assumere solo due valori distinti convenzionalmente detti `true` e `false`, ossia `vero` e `falso`.
- È importante tenere presente che, tradizionalmente,[2] non esiste un tipo booleano in C.
- Quando si dice che la valutazione di una certa espressione in C risulta falsa, si intende per convenzione che il risultato è il numero (intero o in virgola mobile) zero.
- Tutti gli altri valori numerici (positivi o negativi, interi o in virgola mobile) rappresentano convenzionalmente il vero.

[1] Aggettivo derivato dal nome del matematico e logico inglese George Boole, 1815–1864.

[2] In verità, lo standard C99 ha introdotto un tipo booleano, ma non ne parleremo.

Operatori relazionali

Operatore	No. Argomenti	Descrizione
>	2	Maggiore
<	2	Minore
>=	2	Maggiore o uguale
<=	2	Minore o uguale
==	2	Uguale
!=	2	Diverso

Operatori relazionali

Operatore	No. Argomenti	Descrizione
>	2	Maggiore
<	2	Minore
>=	2	Maggiore o uguale
<=	2	Minore o uguale
==	2	Uguale
!=	2	Diverso

Nota. Uno degli errori più comuni di chi comincia a programmare in C è l'uso di = in luogo di ==.

I confronti di uguaglianza si eseguono con ==.

L'operatore = è riservato agli assegnamenti, come vedremo.

Operatori relazionali

Operatore	No. Argomenti	Descrizione
>	2	Maggiore
<	2	Minore
>=	2	Maggiore o uguale
<=	2	Minore o uguale
==	2	Uguale
!=	2	Diverso

Risultato della valutazione. L'espressione relazionale

$$\texttt{a} > \texttt{b}$$

ha valore 1 se la disuguaglianza stretta vale, e 0 altrimenti. Dire che la disuguaglianza 'vale' vuol dire che *vale fra i valori risultanti dalla valutazione di* `a` *e* `b`, *rispettivamente*. Considerazioni analoghe valgono per gli altri operatori.

Operatori relazionali

Operatore	No. Argomenti	Descrizione
>	2	Maggiore
<	2	Minore
>=	2	Maggiore o uguale
<=	2	Minore o uguale
==	2	Uguale
!=	2	Diverso

Esempio. L'espressione $1 - (2 * 3) > -2$ valuta a 0 perché l'espressione $1 - (2 * 3)$ valuta a -5, l'espressione -2 valuta a -2, e $-5 > -2$ è falso.

Operatori relazionali

Operatore	No. Argomenti	Descrizione
>	2	Maggiore
<	2	Minore
>=	2	Maggiore o uguale
<=	2	Minore o uguale
==	2	Uguale
!=	2	Diverso

Esempio. L'espressione $1 - (2 * 3) > -2$ valuta a 0 perché l'espressione $1 - (2 * 3)$ valuta a -5, l'espressione -2 valuta a -2, e $-5 > -2$ è falso. L'espressione $1 - (2 * 3) <= -2$, invece, valuta a 1, preso come *rappresentante canonico* del valore vero: si ricordi però che, come già detto, *qualunque* valore non nullo rappresenta, in C, il vero.

Operatori logici

Operatore	No. Argomenti	Descrizione
&&	2	Congiunzione (AND)
||	2	Disgiunzione inclusiva (OR)
!	1	Negazione (NOT)

Operatori logici

Operatore	No. Argomenti	Descrizione
&&	2	Congiunzione (AND)
||	2	Disgiunzione inclusiva (OR)
!	1	Negazione (NOT)

Risultato della valutazione.

- `a && b` vale 1 se la valutazione di `a` e di `b` dà risultato vero, e vale 0 altrimenti.
- `a || b` vale 1 se la valutazione di almeno uno fra `a` e `b` dà risultato vero, e vale 0 altrimenti.
- `!a` vale 1 se la valutazione di `a` dà risultato falso, e vale 0 altrimenti.

Operatori logici

Operatore	No. Argomenti	Descrizione
&&	2	Congiunzione (AND)
||	2	Disgiunzione inclusiva (OR)
!	1	Negazione (NOT)

Nota. In C, una congiunzione o disgiunzione è valutata da sinistra verso destra. Attenzione, però:

La valutazione si interrompe non appena il risultato si rivela vero o falso.

Si parla in questo caso di valutazione cortocircuitata — in inglese, *short-circuit evaluation*.

Operatori logici

Operatore	No. Argomenti	Descrizione
&&	2	Congiunzione (AND)
||	2	Disgiunzione inclusiva (OR)
!	1	Negazione (NOT)

Nota. In C, una congiunzione o disgiunzione è valutata da sinistra verso destra. Attenzione, però:

La valutazione si interrompe non appena il risultato si rivela vero o falso.

Si parla in questo caso di valutazione cortocircuitata — in inglese, *short-circuit evaluation*. Per esempio, la valutazione di `1||(a+b)` è `1`, indipendentemente dalla valutazione di `a+b`.

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera
(-2<=3.0)+2		

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera
(-2<=3.0)+2	3	

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera
(-2<=3.0)+2	3	vera

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera
(-2<=3.0)+2	3	vera
-3+(-2<=3.0)+2		

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera
(-2<=3.0)+2	3	vera
-3+(-2<=3.0)+2	0	

Esempi

Espressione	Valutazione	Vera o Falsa?
23	23	vera
-23	-23	vera
2.3	2.3	vera
-2.3	-2.3	vera
0.0	0.0	falsa
0	0	falsa
23*-34.45	-792.35	vera
(23*-34.45)*0.0	0.0	falsa
-2>3.0	0	falsa
-2<=3.0	1	vera
(-2<=3.0)+2	3	vera
-3+(-2<=3.0)+2	0	falsa

Dichiarazioni e assegnamenti

In questa parte della lezione, approfondiremo due argomenti solo accennati la volta scorsa: dichiarazioni e assegnamenti.

- Per poter essere usate, le variabili vanno dichiarate.

Dichiarazioni e assegnamenti

In questa parte della lezione, approfondiremo due argomenti solo accennati la volta scorsa: dichiarazioni e assegnamenti.

- Per poter essere usate, le variabili vanno dichiarate.
- Nella prima parte del corso, *dichiareremo sempre le variabili all'interno delle funzioni.*

Dichiarazioni e assegnamenti

In questa parte della lezione, approfondiremo due argomenti solo accennati la volta scorsa: dichiarazioni e assegnamenti.

- Per poter essere usate, le variabili vanno dichiarate.
- Nella prima parte del corso, *dichiareremo sempre le variabili all'interno delle funzioni.*
- Tali variabili sono dette automatiche nel gergo del C, per motivi che vedremo in seguito. In altri linguaggi di programmazione si usa il termine variabile locale, che può anche essere usato nel contesto del linguaggio C.

Dichiarazioni e assegnamenti

In questa parte della lezione, approfondiremo due argomenti solo accennati la volta scorsa: dichiarazioni e assegnamenti.

- Per poter essere usate, le variabili vanno dichiarate.
- Nella prima parte del corso, *dichiareremo sempre le variabili all'interno delle funzioni.*
- Tali variabili sono dette automatiche nel gergo del C, per motivi che vedremo in seguito. In altri linguaggi di programmazione si usa il termine variabile locale, che può anche essere usato nel contesto del linguaggio C.
- Spesso le variabili sono dichiarate *all'inizio* del corpo della funzione cui appartengono, e cioè subito dopo la parentesi graffa che delimita l'inizio del corpo. Ciò non è necessario, però; vedremo più avanti che non è neppure sempre desiderabile.

Nota sullo standard.

Fino allo standard C90, le dichiarazioni delle variabili locali erano ammesse solamente all'inizio di un blocco di codice, un elemento sintattico del sorgente che è delimitato da parentesi graffe, e il cui esempio più ricorrente è il corpo di una funzione. Dallo standard C99 in poi questa restrizione è stata rimossa. Noi ci sentiremo liberi di dichiarare variabili anche nel bel mezzo di un blocco. In laboratorio vedremo esempi di compilazione secondo gli standard C90 e C99.

Nota sullo standard.

Fino allo standard C90, le dichiarazioni delle variabili locali erano ammesse solamente all'inizio di un blocco di codice, un elemento sintattico del sorgente che è delimitato da parentesi graffe, e il cui esempio più ricorrente è il corpo di una funzione. Dallo standard C99 in poi questa restrizione è stata rimossa. Noi ci sentiremo liberi di dichiarare variabili anche nel bel mezzo di un blocco. In laboratorio vedremo esempi di compilazione secondo gli standard C90 e C99.

- La forma sintattica di una dichiarazione di variabile è:

$$\textit{tipo identificatore};$$

Nota sullo standard.

Fino allo standard C90, le dichiarazioni delle variabili locali erano ammesse solamente all'inizio di un blocco di codice, un elemento sintattico del sorgente che è delimitato da parentesi graffe, e il cui esempio più ricorrente è il corpo di una funzione. Dallo standard C99 in poi questa restrizione è stata rimossa. Noi ci sentiremo liberi di dichiarare variabili anche nel bel mezzo di un blocco. In laboratorio vedremo esempi di compilazione secondo gli standard C90 e C99.

- La forma sintattica di una dichiarazione di variabile è:

$$\textit{tipo identificatore};$$

- Gli identificatori sono una categoria lessicale del linguaggio C. Senza scendere nei dettagli, li possiamo pensare come i nomi delle variabili, delle funzioni, e di altri elementi sintattici del codice sorgente.

- Non tutte le sequenze di caratteri sono identificatori validi: per esempio, i nomi di variabile *non devono contenere spazi*, e *non possono cominciare con una cifra*.

- Non tutte le sequenze di caratteri sono identificatori validi: per esempio, i nomi di variabile *non devono contenere spazi*, e *non possono cominciare con una cifra*.
- Occorre anche ricordare che gli identificatori sono "case-sensitive": `pippo`, `Pippo` e `Pipp0` sono tre identificatori (validi) *distinti*.

- Non tutte le sequenze di caratteri sono identificatori validi: per esempio, i nomi di variabile *non devono contenere spazi*, e *non possono cominciare con una cifra*.
- Occorre anche ricordare che gli identificatori sono "case-sensitive": `pippo`, `Pippo` e `PippO` sono tre identificatori (validi) *distinti*.

Identificatore	Valido	Non valido
`x`	✓	
`LaMiaVariabile`	✓	
`mia var`		✓
`mia_var`	✓	
`_x`	✓	
`x1`	✓	
`1x`		✓
`x;y`		✓

- Sintassi alternativa per dichiarazioni multiple:

$$\textit{tipo}\ \textit{identificatore}_1, \ldots, \textit{identificatore}_n\,;$$

- Sintassi alternativa per dichiarazioni multiple:

  $$\textit{tipo identificatore}_1, \ldots, \textit{identificatore}_n;$$

- All'interno di una funzione, una variabile automatica può essere usata *a partire dal punto del codice in cui è dichiarata.*

- Sintassi alternativa per dichiarazioni multiple:

  $$\textit{tipo}\ \textit{identificatore}_1, \ldots, \textit{identificatore}_n;$$

- All'interno di una funzione, una variabile automatica può essere usata *a partire dal punto del codice in cui è dichiarata.* Esempi di dichiarazione:

```
int fuffa, fiffi;
char car;
```

- Sintassi alternativa per dichiarazioni multiple:

  $$\textit{tipo}\ \textit{identificatore}_1, \ldots, \textit{identificatore}_n;$$

- All'interno di una funzione, una variabile automatica può essere usata *a partire dal punto del codice in cui è dichiarata*. Esempi di dichiarazione:

```
int fuffa, fiffi;
char car;
```

- Le variabili possono anche essere inizializzate contestualmente alla loro dichiarazione:

```
int fuffa=-1, fiffi;
char car='A';
```

- Sintassi alternativa per dichiarazioni multiple:

  $$\textit{tipo}\ \textit{identificatore}_1, \ldots, \textit{identificatore}_n;$$

- All'interno di una funzione, una variabile automatica può essere usata *a partire dal punto del codice in cui è dichiarata*. Esempi di dichiarazione:

```
int fuffa, fiffi;
char car;
```

- Le variabili possono anche essere inizializzate contestualmente alla loro dichiarazione:

```
int fuffa=-1, fiffi;
char car='A';
```

- Le variabili dichiarate ma non inizializzate hanno valore indeterminato secondo lo standard C.

Dichiarazioni e assegnamenti

- Per attribuire un valore a una variabile si usa un'assegnazione (o un assegnamento).

Dichiarazioni e assegnamenti

- Per attribuire un valore a una variabile si usa un'assegnazione (o un assegnamento).
- Sintassi:

$$variabile = espressione;$$

Dichiarazioni e assegnamenti

- Per attribuire un valore a una variabile si usa un'assegnazione (o un assegnamento).
- Sintassi:

$$variabile = espressione;$$

- Semantica (prima approssimazione): Si valuta l'`espressione`, e si assegna il risultato della valutazione a *variabile*. Dopo l'esecuzione dell'istruzione, *variabile* rappresenta dunque il valore risultante dalla valutazione del membro destro.

Dichiarazioni e assegnamenti

- Per attribuire un valore a una variabile si usa un'assegnazione (o un assegnamento).
- Sintassi:

$$variabile = espressione;$$

- Semantica (prima approssimazione): Si valuta l'`espressione`, e si assegna il risultato della valutazione a *variabile*. Dopo l'esecuzione dell'istruzione, *variabile* rappresenta dunque il valore risultante dalla valutazione del membro destro.
- Nota bene. (1) Il vecchio valore contenuto nella variabile è perduto dopo l'assegnamento. (2) Torneremo sulla semantica degli assegnamenti; la descrizione qui sopra è una prima approssimazione che per il momento è sufficiente.

Dichiarazioni e assegnamenti

- Per attribuire un valore a una variabile si usa un'assegnazione (o un assegnamento).
- Sintassi:

$$variabile = espressione;$$

- Semantica (prima approssimazione): Si valuta l'`espressione`, e si assegna il risultato della valutazione a *variabile*. Dopo l'esecuzione dell'istruzione, *variabile* rappresenta dunque il valore risultante dalla valutazione del membro destro.
- Nota bene. (1) Il vecchio valore contenuto nella variabile è perduto dopo l'assegnamento. (2) Torneremo sulla semantica degli assegnamenti; la descrizione qui sopra è una prima approssimazione che per il momento è sufficiente.
- L'inizializzazione è semplicemente un caso particolare dell'assegnamento.

```
int fuffa=23, fiffi=-1;
fuffa=fiffi+1;
```

- Dopo l'esecuzione, `fuffa` vale `0` e `fiffi` vale `-1`.

```
int fuffa=23, fiffi=-1;
fuffa=fiffi+1;
```

- Dopo l'esecuzione, `fuffa` vale `0` e `fiffi` vale `-1`.
- Se il valore iniziale 23 di `fuffa` aveva un profondo significato nascosto, tanto peggio: dopo l'esecuzione del frammento di codice, quel valore è andato perduto.

```
int fuffa=23, fiffi=-1;
fuffa=fiffi+1;
```

- Dopo l'esecuzione, `fuffa` vale `0` e `fiffi` vale `-1`.
- Se il valore iniziale 23 di `fuffa` aveva un profondo significato nascosto, tanto peggio: dopo l'esecuzione del frammento di codice, quel valore è andato perduto.

Un esercizio classico

Scrivere il codice che scambia i valori di due variabili intere a e b.

Nel risolvere questo esercizio vedremo in dettaglio cosa succede nella memoria centrale del calcolatore durante l'esecuzione di un assegnamento.

Un esercizio classico

Scrivere il codice che scambia i valori di due variabili intere `a` e `b`.

Un esercizio classico

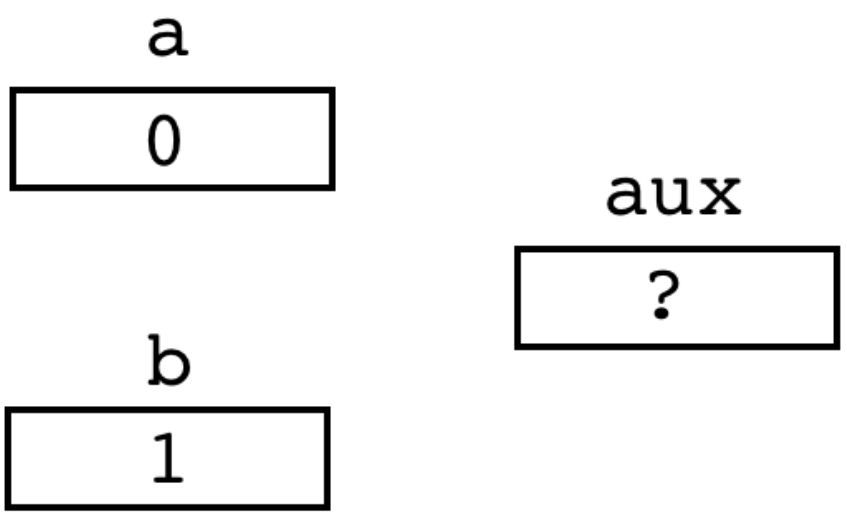
Scrivere il codice che scambia i valori di due variabili intere a e b.

scambia.c

```c
#include <stdio.h>

int main(void)
{
        int a=0, b=1, aux; //E' necessaria una variabile ausiliaria

        printf("a=%d, b=%d.\n",a,b); //Prima

        aux=a;  //...salva il valore di a in aux
        a=b;    //...il valore di a e' perduto
        b=aux;  //...ma una copia era stata salvata in aux

        printf("a=%d, b=%d.\n",a,b); //Dopo

        return 0;
}
```

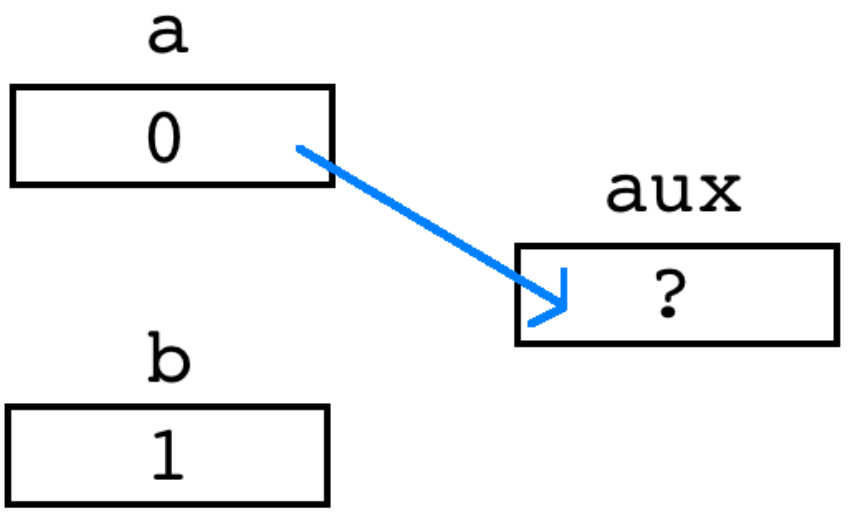
Semantica operazionale degli assegnamenti

```
5       int a=0, b=1, aux; //E' necessaria una variabile ausiliaria
```



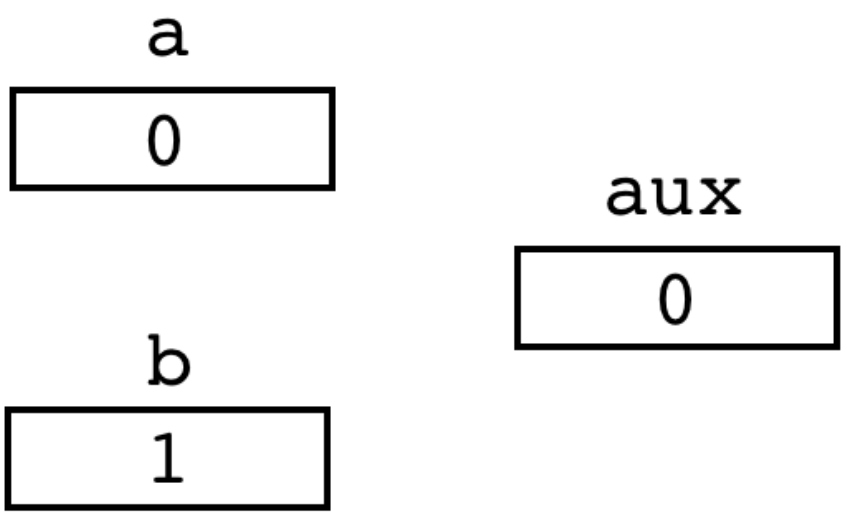
Semantica operazionale degli assegnamenti

```
9        aux=a;  //...salva il valore di a in aux
```



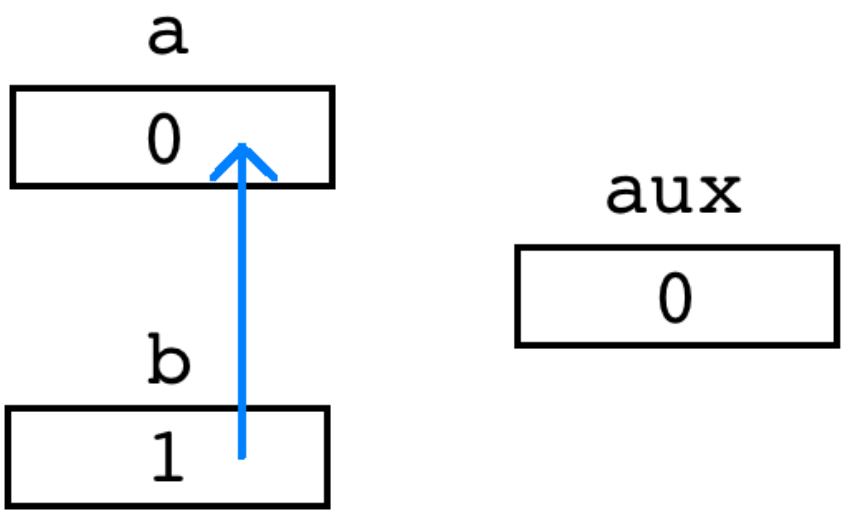
Semantica operazionale degli assegnamenti

```
9        aux=a;  //...salva il valore di a in aux
```



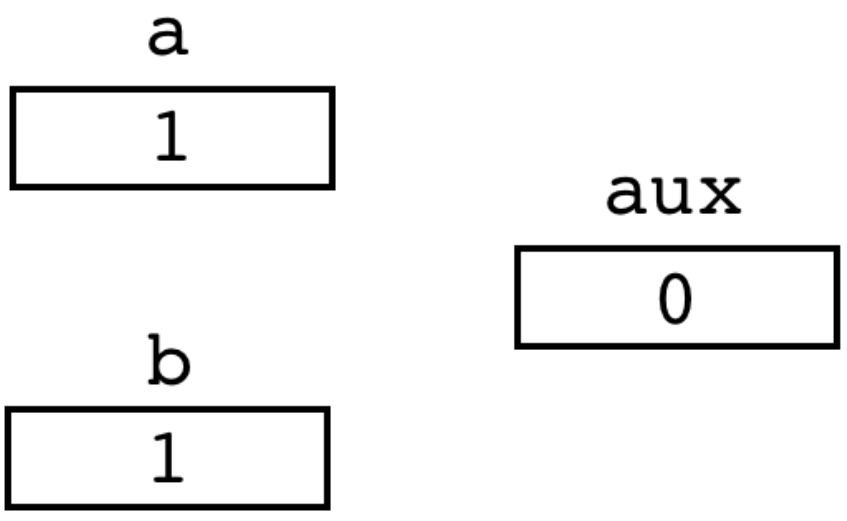
Semantica operazionale degli assegnamenti

```
10          a=b;    //...il valore di a e' perduto
```



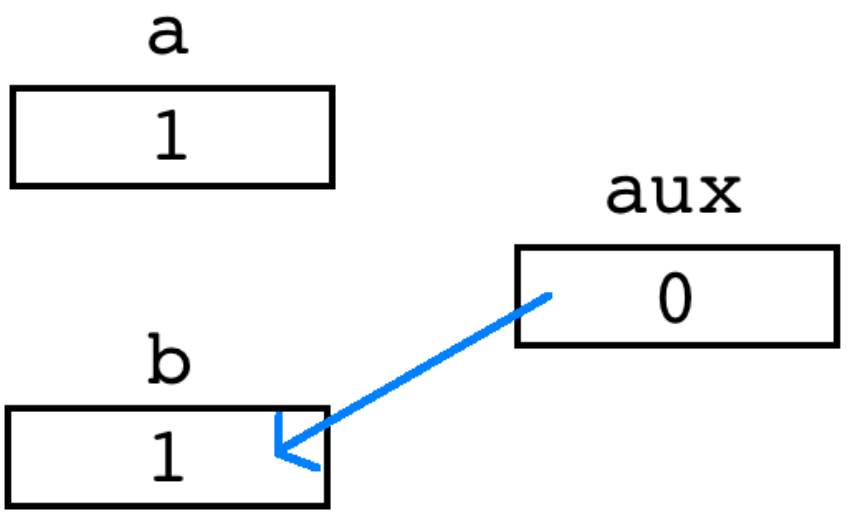
Semantica operazionale degli assegnamenti

```
10          a=b;    //...il valore di a e' perduto
```



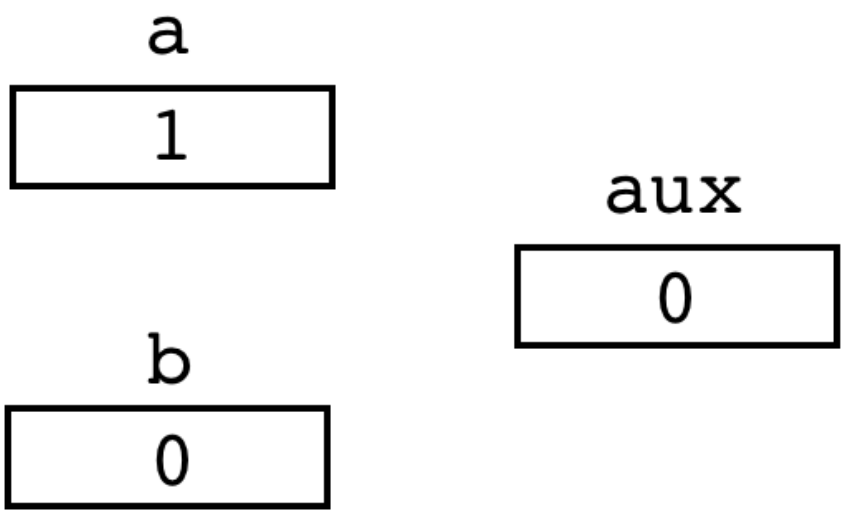
Semantica operazionale degli assegnamenti

```
11        b=aux;  //...ma una copia era stata salvata in aux
```



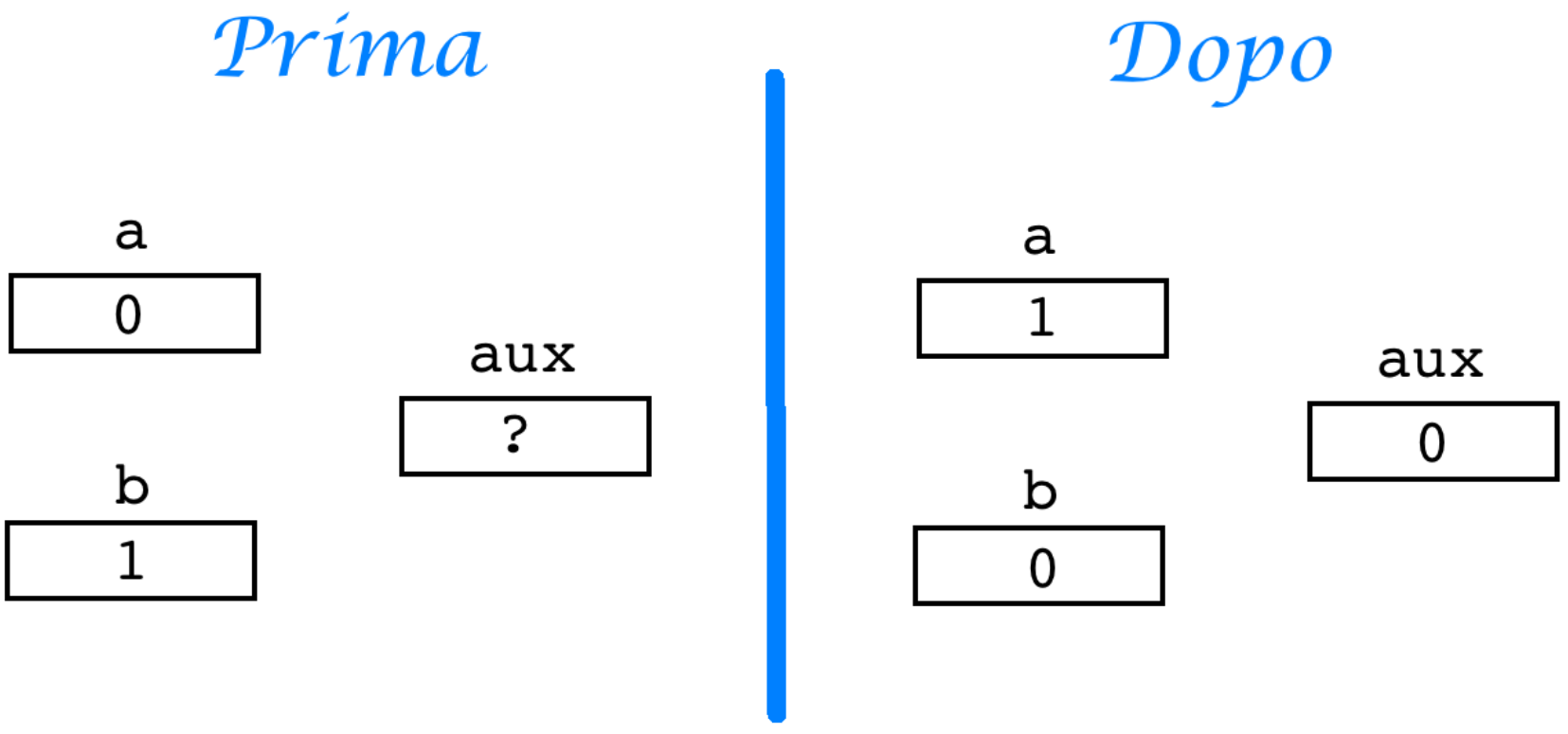
Semantica operazionale degli assegnamenti

```
11        b=aux;  //...ma una copia era stata salvata in aux
```



Semantica operazionale degli assegnamenti

```
9        aux=a;  //...salva il valore di a in aux
10       a=b;    //...il valore di a e' perduto
11       b=aux;  //...ma una copia era stata salvata in aux
```



Il flusso del controllo

- Durante l'esecuzione di un programma, ad ogni istante di tempo la macchina è impegnata ad eseguire una certa istruzione del programma stesso. Si dice che il controllo del programma è assegnato a quell'istruzione (in quell'istante). La sequenza di istruzioni eseguite è il flusso del controllo, in inglese *control flow* o *flow of control*.

Il flusso del controllo

- Durante l'esecuzione di un programma, ad ogni istante di tempo la macchina è impegnata ad eseguire una certa istruzione del programma stesso. Si dice che il controllo del programma è assegnato a quell'istruzione (in quell'istante). La sequenza di istruzioni eseguite è il flusso del controllo, in inglese *control flow* o *flow of control*.
- In assenza di indicazioni diverse, il flusso del controllo procede dalla prima istruzione del sorgente all'ultima, in modo lineare.

Il flusso del controllo

- Durante l'esecuzione di un programma, ad ogni istante di tempo la macchina è impegnata ad eseguire una certa istruzione del programma stesso. Si dice che il controllo del programma è assegnato a quell'istruzione (in quell'istante). La sequenza di istruzioni eseguite è il flusso del controllo, in inglese *control flow* o *flow of control*.
- In assenza di indicazioni diverse, il flusso del controllo procede dalla prima istruzione del sorgente all'ultima, in modo lineare.
- Le istruzioni di un linguaggio di programmazione che servono a modificare il flusso del controllo di un programma sono dette istruzioni relative al flusso del controllo, in inglese *control flow statements*.

- Il linguaggio C fornisce diverse istruzioni relative al flusso del controllo. Cominceremo a discuterne due, che appartengono, rispettivamente, alle categorie astratte delle istruzioni di iterazione e selezione. Si tratta delle istruzioni:

  `while...`, e

  `if...else...`

- Il linguaggio C fornisce diverse istruzioni relative al flusso del controllo. Cominceremo a discuterne due, che appartengono, rispettivamente, alle categorie astratte delle istruzioni di iterazione e selezione. Si tratta delle istruzioni:

  `while...`, e

  `if...else...`

- Informalmente, la prima istruzione permette di implementare delle *iterazioni*, dette anche *cicli* o *loop* — permette cioè di ripetere l'esecuzione di una porzione di codice sorgente fino a quando una data condizione non diventi falsa.

- Il linguaggio C fornisce diverse istruzioni relative al flusso del controllo. Cominceremo a discuterne due, che appartengono, rispettivamente, alle categorie astratte delle istruzioni di iterazione e selezione. Si tratta delle istruzioni:

  `while...`, e

  `if...else...`

- Informalmente, la prima istruzione permette di implementare delle *iterazioni*, dette anche *cicli* o *loop* — permette cioè di ripetere l'esecuzione di una porzione di codice sorgente fino a quando una data condizione non diventi falsa.
- La seconda istruzione, invece, permette di eseguire una o l'altra di due porzioni di codice a seconda del verificarsi o meno di una data condizione.

Il flusso del controllo: `while`

- Sintassi.

`while` (*espressione*)

istruzione

Il flusso del controllo: `while`

- Sintassi.

`while` (*espressione*)

istruzione

- Semantica.
  1. Si valuta *espressione*.

Il flusso del controllo: `while`

- Sintassi.

`while` (*espressione*)

istruzione

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue *istruzione*, e si torna al punto 1.

Il flusso del controllo: `while`

- Sintassi.

`while` (*espressione*)

istruzione

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue *istruzione*, e si torna al punto 1.
  3. Questo ciclo prosegue fino a quando la valutazione di *espressione* risulta in un valore `falso`: in questo caso, l'esecuzione prosegue dal punto seguente a *istruzione*.

Un esempio: while

while.c

```c
/* Un esempio di ciclo while */

#include <stdio.h>

int main(void)
{
        int base=5, esp=6, ris=1; // Precondizioni: base > 0, esp >= 0, ris = 1

        printf("Calcolo %d elevato alla %d: \n",base,esp);

        while (esp)
        {                                       //Questo intero blocco...
                ris=ris*base;
                esp=esp-1;
                printf("ris=%d\tbase=%d\tesp=%d\n",ris,base,esp);
        }                                       //...conta come una istruzione.

        printf("%d.\n",ris);

        return 0;
}
```

(*Esecuzione dell'esempio.*)

Il flusso del controllo: `if...else...`

- Sintassi.

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

Nota. La clausola `else` è facoltativa.

Il flusso del controllo: `if...else...`

- Sintassi.

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

  Nota. La clausola `else` è facoltativa.

- Semantica.
  1. Si valuta *espressione*.

Il flusso del controllo: `if...else...`

- Sintassi.

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

  Nota. La clausola `else` è facoltativa.

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue $istruzione_1$, e si prosegue con la prima istruzione dopo l'istruzione `if...else...`.

Il flusso del controllo: `if...else...`

- Sintassi.

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

  Nota. La clausola `else` è facoltativa.

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue $istruzione_1$, e si prosegue con la prima istruzione dopo l'istruzione `if...else...`.
  3. Se il risultato è `falso`, ed è presente la clausola `else`, si esegue $istruzione_2$, e si prosegue con la prima istruzione dopo l'istruzione `if...else...`.

Il flusso del controllo: `if...else...`

- Sintassi.

```
if (espressione)
    istruzione1
else
    istruzione2
```

  Nota. La clausola `else` è facoltativa.

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue $istruzione_1$, e si prosegue con la prima istruzione dopo l'istruzione `if...else...`.
  3. Se il risultato è `falso`, ed è presente la clausola `else`, si esegue $istruzione_2$, e si prosegue con la prima istruzione dopo l'istruzione `if...else...`.
  4. Se il risultato è `falso`, e non è presente la clausola `else`, si prosegue con la prima istruzione dopo l'istruzione `if...`.

Un esempio: if

if.c

```c
/* Un esempio di selezione con l'istruzione if */

#include <stdio.h>
#include <ctype.h>          //Per la funzione int isdigit(int c) della libreria

int main(void)
{
        char c;

        printf("Digita un carattere: ");
        c=getchar();

        if ( isdigit(c) )   //Vera se c e' una cifra. Equivale a isdigit(c) != 0
                printf("Hai digitato una cifra.\n");
        else
                printf("Non hai digitato una cifra.\n");

        return 0;
}
```

(*Esecuzione dell'esempio.*)

Euclide

Euclide.c

```c
/* L'algoritmo euclideo delle sottrazioni successive */
// Input: due interi a,b > 0, a>=b.
// Output: mcd(a,b)

#include <stdio.h>

int main(void)
{
        int a=24;
        int b=16;

        printf("Il mcd di %d e %d e': ",a,b);

        while (a != b)
        {
                if (a>b)
                        a=a-b;
                else
                        b=b-a;
        }
        printf("%d.\n", a);
        return 0;
}
```

(*Esecuzione dell'algoritmo euclideo.*)

Il flusso del controllo: la variante do-while di while

- Sintassi.

`while` (*espressione*)

istruzione

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue *istruzione*, e si torna al punto 1.
  3. Questo ciclo prosegue fino a quando la valutazione di *espressione* risulta in un valore `falso`: in questo caso, l'esecuzione prosegue dal punto seguente a *istruzione*.

Il flusso del controllo: do-while

- Sintassi.

do *istruzione*

while (*espressione*)

Il flusso del controllo: do-while

- Sintassi.

  `do` *istruzione*

  `while (`*espressione*`)`

- Semantica.
  1. Si esegue *istruzione*.

Il flusso del controllo: do-while

- Sintassi.

  `do` *istruzione*

  `while (`*espressione*`)`

- Semantica.
  1. Si esegue *istruzione*.
  2. Si valuta *espressione*.

Il flusso del controllo: do-while

- Sintassi.

  `do` *istruzione*

  `while (`*espressione*`)`

- Semantica.
  1. Si esegue *istruzione*.
  2. Si valuta *espressione*.
  3. Se il risultato è `vero`, si torna al punto 1.

Il flusso del controllo: do-while

- Sintassi.

  `do` *istruzione*

  `while (`*espressione*`)`

- Semantica.
  1. Si esegue *istruzione*.
  2. Si valuta *espressione*.
  3. Se il risultato è `vero`, si torna al punto 1.
  4. Questo ciclo prosegue fino a quando la valutazione di *espressione* risulta in un valore `falso`: in questo caso, l'esecuzione prosegue dal punto seguente a *espressione*.

Uso `while` oppure `do-while`?

- `while` si usa quando si vuole ripetere un blocco di codice n volte, con $n \geqslant 0$ — quindi, in qualche esecuzione, il blocco potrebbe anche non essere eseguito affatto. (Ciò accade quando già la prima valutazione di *espressione* risulta in `falso`).

Uso `while` oppure `do-while`?

- `while` si usa quando si vuole ripetere un blocco di codice n volte, con $n \geqslant 0$ — quindi, in qualche esecuzione, il blocco potrebbe anche non essere eseguito affatto. (Ciò accade quando già la prima valutazione di *espressione* risulta in `falso`).
- `do-while` si usa quando si vuole ripetere un blocco di codice n volte, con $n > 0$ — quindi, in tutte le esecuzioni, il blocco sarà eseguito almeno una volta.

Uso `while` oppure `do-while`?

- `while` si usa quando si vuole ripetere un blocco di codice n volte, con $n \geqslant 0$ — quindi, in qualche esecuzione, il blocco potrebbe anche non essere eseguito affatto. (Ciò accade quando già la prima valutazione di *espressione* risulta in `falso`).
- `do-while` si usa quando si vuole ripetere un blocco di codice n volte, con $n > 0$ — quindi, in tutte le esecuzioni, il blocco sarà eseguito almeno una volta.
- Esempio. *Spedisci (periodicamente) una copia della missiva fino a quando non accusi ricevuta.*

Uso `while` oppure `do-while`?

- `while` si usa quando si vuole ripetere un blocco di codice n volte, con $n \geqslant 0$ — quindi, in qualche esecuzione, il blocco potrebbe anche non essere eseguito affatto. (Ciò accade quando già la prima valutazione di *espressione* risulta in `falso`).
- `do-while` si usa quando si vuole ripetere un blocco di codice n volte, con $n > 0$ — quindi, in tutte le esecuzioni, il blocco sarà eseguito almeno una volta.
- Esempio. *Spedisci (periodicamente) una copia della missiva fino a quando non accusi ricevuta.*

```
do spedisci copia
while (non accusi ricevuta)
```

Uso `while` oppure `do-while`?

- `while` si usa quando si vuole ripetere un blocco di codice n volte, con $n \geqslant 0$ — quindi, in qualche esecuzione, il blocco potrebbe anche non essere eseguito affatto. (Ciò accade quando già la prima valutazione di *espressione* risulta in `falso`).
- `do-while` si usa quando si vuole ripetere un blocco di codice n volte, con $n > 0$ — quindi, in tutte le esecuzioni, il blocco sarà eseguito almeno una volta.
- Esempio. *Aggiungi acqua fino a quando il secchio non è pieno.*

Uso `while` oppure `do-while`?

- `while` si usa quando si vuole ripetere un blocco di codice n volte, con $n \geqslant 0$ — quindi, in qualche esecuzione, il blocco potrebbe anche non essere eseguito affatto. (Ciò accade quando già la prima valutazione di *espressione* risulta in `falso`).
- `do-while` si usa quando si vuole ripetere un blocco di codice n volte, con $n > 0$ — quindi, in tutte le esecuzioni, il blocco sarà eseguito almeno una volta.
- Esempio. *Aggiungi acqua fino a quando il secchio non è pieno.*

`while` (*secchio non pieno*)

aggiungi acqua