# Espressioni e funzioni matematiche

Ver. 3

© 2021 - Claudio Fornaro - Corso di programmazione in C (C89)

# Espressioni numeriche

- Sono composte da operatori, variabili, costanti e funzioni, producono un valore

```
Delta = b*b-4*a*c;
```

- Gli operatori matematici sono:

| | | |
|---|---|---|
| `+` | somma | `x = a + b;` |
| `-` | sottrazione | `x = a - b;` |
| `*` | moltiplicazione | `x = a * b;` |
| `/` | divisione | `x = a / b;` |
| `%` | resto della divisione intera | `x = a % b;` |

- Il calcolo della divisione e del resto di un valore per zero ha risultato indefinito (“undefined-behavior”), in genere dà errore

# Espressioni numeriche

- La *divisione* di due valori interi dà un risultato intero **troncato** della parte frazionaria e non arrotondato all'intero più vicino (8/3 che vale 2.66666… dà 2 e non 3)
- Se gli operandi sono discordi il risultato negativo viene troncato *tipicamente* al numero intero subito più a destra (ossia –5/4, che vale –1.25, dà –1), ma lo Standard C89 non lo specifica e a seconda del compilatore usato il troncamento potrebbe invece dare il valore negativo subito *a sinistra* del valore dato: (int)(–1.25) = –2

# Espressioni numeriche

- Il *resto della divisione* può essere calcolato solo tra due valori di tipo intero (non floating-point, per i quali esiste la funzione `fmod`)
- Se gli operandi sono discordi, ***tipicamente*** il calcolo viene fatto come se i numeri fossero positivi e il segno è quello del dividendo (–9%7= –2); ma lo Standard C89 non lo specifica e a seconda del compilatore –9%7 può dare o –2 calcolando –(9%7) o +5 calcolando –(9%7)+9

# Precedenza degli operatori

- Le regole di *precedenza* (o *priorità*) specificano in quale ordine vengono eseguiti i calcoli
- Raggruppati in livelli di priorità decrescente:
  1. `+ -` → *segno*
  2. `()`
  3. `* / %`
  4. `+ -` → *somma e sottrazione*
- Esempi

```
x = a+b*c;
```
→ *prima la moltiplicazione*

```
x = (a+b)*c;
```
→ *prima la somma*

```
x = a - -b;
```
→ `a - (-b)`

sottrazione (primo `-`) — segno (secondo `-`)

# Associatività degli operatori

- Le regole di *associatività* specificano in quale ordine vengono eseguiti i calcoli con operatori *aventi lo stesso livello di precedenza*
- Per gli operatori matematici l'associatività è sempre da sinistra a destra

```
x=a+b+c;       →    x=(a+b)+c;
x=a+b-c+d;     →    x=(((a+b)-c)+d);
x=a*b/c;       →    x=(a*b)/c;
x=a+b+c*d;     →    x=((a+b)+(c*d));
```

# Espressioni numeriche

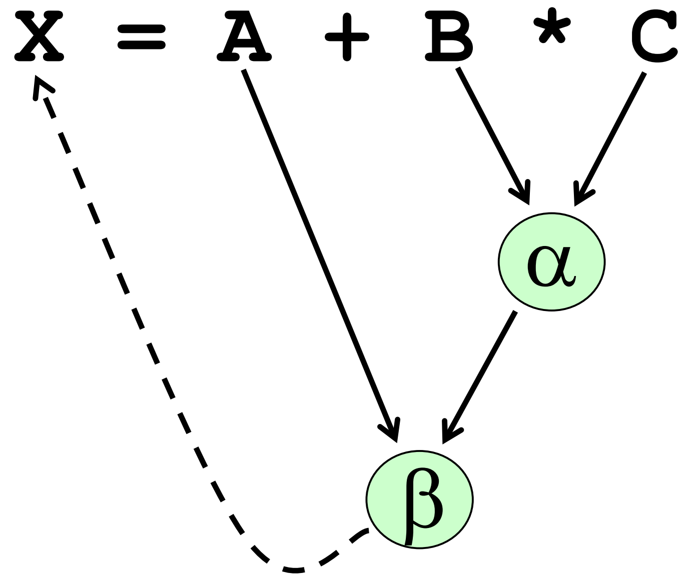
## Operandi dello stesso tipo

- Le operazioni matematiche possono essere eseguite solo tra due operandi dello stesso tipo base (`int`, `long`, `float`, `double`, `long double`), gli interi devono essere entrambi `signed` o `unsigned`
- I risultati intermedi dei calcoli vengono memorizzati in *variabili temporanee* (senza nome) *dello stesso tipo degli operandi*
- Le variabili temporanee vengono rimosse automaticamente dalla memoria dopo essere state utilizzate

# Espressioni numeriche

## Operandi dello stesso tipo






α e β sono le variabili temporanee

# Espressioni numeriche

## Operandi di tipo diverso

- Le operazioni tra operandi di tipo diverso non possono essere immediatamente calcolate, ma ci si deve ricondurre al caso di tipi uguali
- Vengono quindi applicate le ***promozioni:*** il *valore* dell'operando con il tipo inferiore (meno capiente) viene convertito nel tipo dell'altro
- Ad esempio:
  - `int + long` → **long** + `long`
  - `double + float` → `double +` **double**
  - `int + float` → **float** + `float`

# Espressioni numeriche
## Promozioni

- La promozione crea automaticamente una *variabile temporanea* del tipo più capiente dei due *preservandone il valore numerico*
- Ad es. il valore **3** di tipo `int` (in Complemento a 2) potrebbe essere convertito nel valore **3.0** di tipo `double` (in floating-point IEEE 754 DP)
- Un valore *grande* potrebbe essere convertito al tipo più capiente introducendo eventualmente un'approssimazione, ad es. un `long` a 32 bit pari a 1234567999 equivale ad un `float` a 32 bit approssimato a circa 1.234567e9, ossia 1234567000

# Espressioni numeriche



## Promozioni

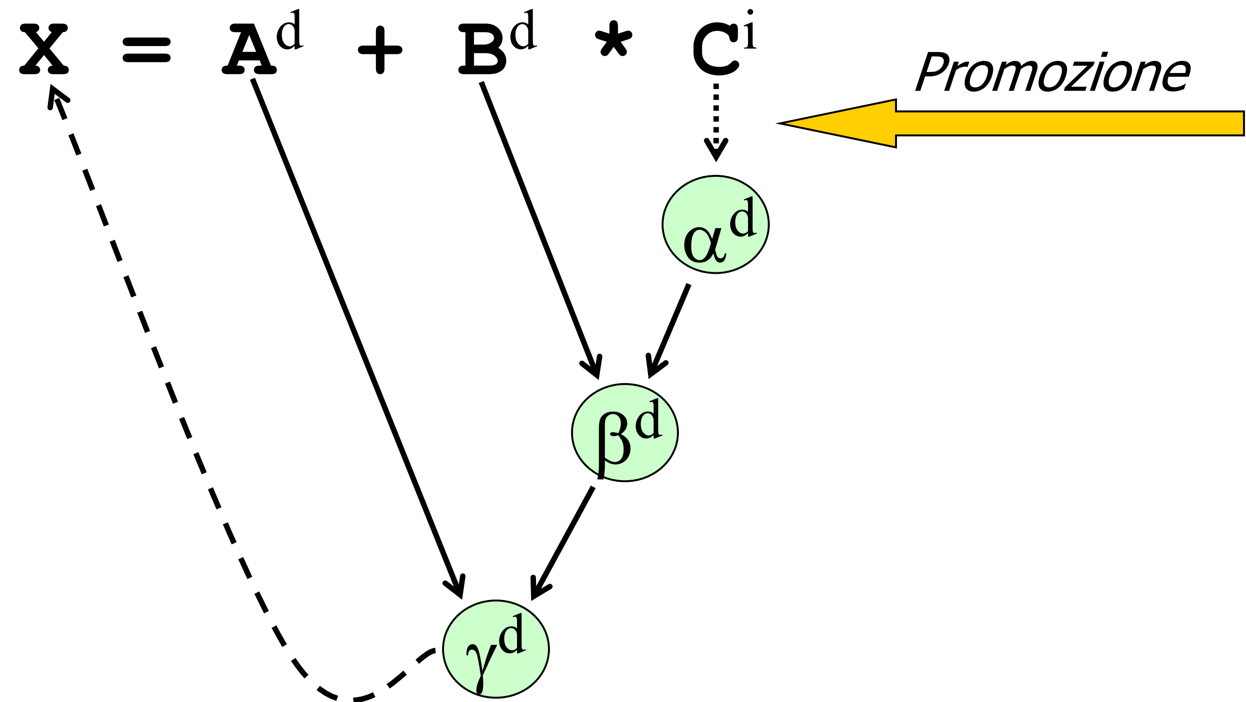
- Nel caso di valori floating-point, la promozione prevede le "*usual arithmetic conversions*":
  `float` < `double` < `long double`
- Nel caso di valori interi, si hanno le "integral promotions":
  `int` < `unsigned int` < `long` < `unsigned long int`
- Nel caso di interi dello stesso tipo ma uno `signed` e uno `unsigned`, l'intero `signed` è convertito in `unsigned` e nel caso di valori negativi il risultato è **errato** (ad es. gli stessi bit sono considerati come valore `unsigned`)

# Espressioni numeriche
## Promozioni integrali

- Il C89 attua le *promozioni integrali* per le quali nelle espressioni matematiche i valori dei tipi `char` e `short int`, i campi di bit e gli elementi delle `enum` sono sempre convertiti:
  - in `int` se questo tipo può rappresentare i valori originali
  - in `unsigned int` altrimenti
- Ad es. una somma tra `short` avviene solo dopo aver promosso gli operandi a `int`

# Espressioni numeriche
## Operandi di tipo diverso



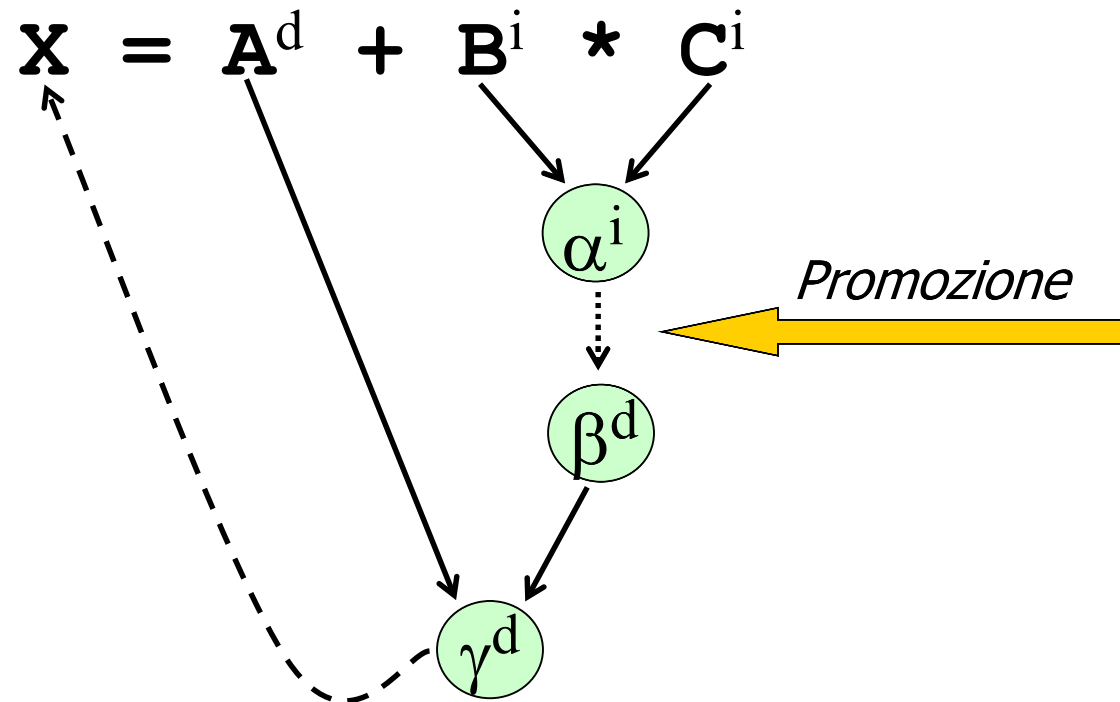



In questi schemi la lettera d indica che la variabile (anche temporanea) o il valore è di tipo `double`, i di tipo `int`

# Espressioni numeriche

## Operandi di tipo diverso

- La promozione avviene *solo nel momento in cui è necessaria* per proseguire il calcolo




Il calcolo tra **B** e **C** non richiede alcuna promozione, a differenza di quello tra **A** ed α

# Espressioni numeriche

## Operandi di tipo diverso

- **Attenzione**
  Supponiamo gli `int` siano su 16 bit (quindi max 32767) e i `long` su 32 bit (quindi max 2 miliardi). Il codice seguente dà risultato sbagliato (overflow) nel prodotto:

```
int a, b;
long c;
a = 30000;
b = 2;
c = a * b;
```

Perché?

# Espressioni numeriche

## Operandi di tipo diverso

- Risposta:
  la variabile intermedia $\alpha$ è di tipo `int` e un `int` di 16 bit non può contenere 60000
- Per risolvere il problema serve che la variabile intermedia sia di tipo `long`:
  - o definendo almeno una delle variabili di tipo `long` (perché l'altra verrebbe promossa automaticamente)
  - oppure richiedendo esplicitamente la promozione a `long` di almeno una delle variabili
- La seconda soluzione è migliore: non richiede di cambiare il tipo delle variabili (è il risultato a non essere rappresentabile, non gli operandi)

# Conversione di tipo - cast

- L'operatore di *cast* produce una *variabile temporanea* del tipo indicato contenente il valore originale convertito nel nuovo tipo (si noti che la variabile originaria resta intatta)

  (*tipo*) *espressione*
- Esempio
```
c = (long)a;
```
- Il cast ha priorità maggiore degli operatori matematici per cui se si vuole convertire il risultato dell'espressione è necessario racchiuderla tra parentesi:
```
c = (long)(a*b+c);
```

# Conversione di tipo - cast

- L’operatore di cast opera in modo simile alla promozione, salvo che:
  - la *promozione* è una conversione automatica, il *cast* è una conversione richiesta dal programmatore
  - nella *promozione* il tipo di dato della variabile temporanea viene determinato dal compilatore, nel *cast* è indicato esattamente dal programmatore
- La conversione con cast preserva il valore numerico, salvo avere gli stessi problemi della promozione (di approssimazione e di conversione di valori `signed` in `unsigned`)
- Il cast può essere applicato solo a oggetti scalari (es. non a vettori e `struct`)

# Conversione di tipo - cast

- Soluzioni dell'overflow in `c=a*b`:

  `c = (long)a * b;` → *cast applicato ad* `a`
  `b` *è promosso a* `long`

  `c = a * (long)b;` → *cast applicato a* `b`
  `a` *è promosso a* `long`

  `c = (long)a * (long)b;`

  `c = (long)(`**`a*b`**`);` → *INUTILE!*

  Nell'ultimo caso il cast viene applicato *al risultato dell'operazione*, ma è l'operazione stessa a dare problemi, non l'assegnazione alla variabile `c`

# Conversione di tipo - cast

- La conversione da un tipo più ampio a uno meno ampio (*downgrade*) *può* eliminare bit significativi → *possibile errore di conversione*

  Ad es. se gli `int` sono di 16 bit (max=32767):

  `long x = 123456;` → >max quindi `long`

  `int y = (int)x;`

  il cast non preserva il valore di `x` perché troppo grande per un `int`
- La conversione di un valore floating-point in intero avviene con *troncamento* della parte frazionaria, inoltre la parte intera potrebbe non essere rappresentabile dal tipo intero

# Conversioni nelle assegnazioni

- Quando un valore di un tipo viene assegnato a una variabile di un altro tipo (questo include anche l'*inizializzazione* e il *passaggio come parametro* a una funzione), vi è una implicita conversione del valore al tipo di questa

```
int x = 12;
long y;
y = x;
```

- Essendo un cast (implicito perché non indicato dal programmatore, ma vale tutto quanto già scritto per i cast), quando è possibile l'assegnazione cerca di preservare il valore numerico (3 intero ←→ 3.0 floating point)

# Conversioni nelle assegnazioni

- In caso di *downgrade* i compilatori effettuano la conversione, ma danno un ***warning*** *per* segnalare che quell'operazione *potrebbe* dare problemi: il compilatore non può sapere quali valori verranno utilizzati al run-time e se questi saranno compatibili o no con il tipo della variabile a cui assegnarli
- Se si sa che i valori saranno compatibili con il tipo della variabile da assegnare, per sopprimere il warning:
  - per le variabili si usa un cast: `x = (int)y;`
  - per le costanti usare sempre i suffissi:
    `float z = 3.1F;` → altrimenti 3.1 è un `double` che viene convertito in `float` perdendo precisione

# Conversioni nelle assegnazioni

- Esempi (con `long` di 32 bit e `int` di 16 bit)

```
int a, b=12;          → OK (12 è un int)
long l=123456;        → OK (suffisso non necessario)
double e=3.0;         → OK (3.0 è un double)
float g = 9.1;        → Warning (g approssimato)
float h = 9.0;        → Possibile Warning (ma OK)
float j = 9.1F;       → OK (utilizzare sempre suffisso)
a = e;                → Warning (a vale 3, ma OK)
a = (int)e;           → No warning (a vale 3, meglio)
a = (int)g;           → No warning (a vale 9, meglio)
e = b;                → No warning (e vale 12.0)
a = l;                → Warning (a vale ???, NO)
a = (int)l;           → No warning (a vale ???, NO)
g = e;                → Warning (g approssimato, NO)
```

# Conversioni nei confronti

- Gli operandi di un’operazione di confronto le stesse regole di conversione delle promozioni
- Si faccia attenzione al caso di confronto di interi dello stesso tipo ma uno `signed` *negativo* e uno `unsigned`, come già scritto la conversione del `signed` non è corretta e il confronto dà risultato **errato**
- In genere il compilatore segnala il confronto tra valori con segno e senza segno
- Sta al programmatore prevedere se il problema sussiste e risolverlo opportunamente

# Conversioni nei confronti

- Esempio (supponendo `int` di 16 bit)
  ```
  int x = -1;
  unsigned int y = 2;
  ```
  Il confronto `x > y` è ovviamente FALSO, ma avviene la *promozione* di `x` a `unsigned int`
- Il valore –1 in Compl. a 2 è composto da 16 cifre pari a 1 (1111111111111111) che considerato `unsigned` vale 65535, per cui il risultato di `x > y` darebbe VERO
- Soluzione: applicare un cast appropriato, <u>in questo caso</u> è sufficiente: `x > (int)y`
  in altri serve un tipo `signed` più capiente

# Overflow e Underflow

- Nella valutazione delle espressioni, la gestione dell'overflow, della divisione per 0 e delle altre *eccezioni* (errori al run-time) non è definita dal linguaggio (*undefined behavior* ), ma è dipendente dal compilatore (*implementation dependent* ): in alcuni sistemi vengono ignorati, in altri rilevati
- *Tipicamente* l'overflow prodotto da:
  - operazioni su valori interi: → ignorato
  - operazioni su valori floating-point: → valore speciale INF (infinito, può essere +INF o –INF)
  - divisione per 0: ferma il programma

# Overflow e Underflow

- *Per i valori senza segno,* in caso di overflow il C ha invece un comportamento ben definito: richiede che i compilatori seguano le leggi dell'aritmetica modulo $2^n$ (scarta il riporto):
  - es. su 8 bit: 255+1=0
    infatti 11111111+1=00000000
  - es. su 8 bit: 0–1=255 )
- Quando si sommano due numeri di tipo Floating-Point, se il rapporto tra i due è $<2^{23}$ (IEEE-P754 SP) o $<2^{52}$ (IEEE-P754 DP) si ha un *Underflow*, ossia il minore dei due diventa pari a 0

# l-value e r-value

- I termini *l-value* e *r-value* (o *lvalue* e *rvalue*) derivano dall'espressione di assegnazione E1=E2, L sta per "left", R sta per "right":
  - un (*modifiable*) *l-value* è un "qualcosa" (ad es. una variabile, ma più in generale un'espressione) che si può mettere a sinistra del segno di '=' e quindi è assegnabile con un valore (identifica una zona di memoria indirizzabile)
  - un *r-value* è il risultato di un espressione che può essere messa a destra del segno di '=', può essere una variabile, una costante, il risultato di un calcolo o di un cast, ecc. lo standard preferisce il termine "valore di un espressione"

# Operatori ++ e ––

- Incrementano/decrementano di 1 una variabile intera o floating-point:
  - `++a` incrementa `a` di 1 *prima* che `a` venga utilizzata nel calcolo (*incremento prefisso*)
  - `a++` incrementa `a` di 1 *dopo* che `a` è stata utilizzata nel calcolo (*incremento postfisso*)
  - `--a` e `a--` analogamente decrementano `a` di 1
- Esempi:
  - `a = 5;`
    `x = ++a;`
    ora `a` vale 6 e `x` vale 6
  - `a = 5;`
    `x = a++;`
    ora `a` vale 6 e `x` vale 5

# Operatori ++ e --

- Hanno priorità maggiore degli operatori matematici
- Utilizzabili solo con *modifiable l-value:* ad es. non si possono applicare al risultato di un calcolo (*r-value*)

  `x=(a+1)++;` → *ERRORE*
- Attenzione
  Le variabili usate con un operatore `++` o `--` non possono apparire più di una volta nella stessa espressione (da inizio al ; finale)

  `x = i * i++;` ← *Errore*

  `i = 2 * i++;` ← *Errore*

# Operatori di assegnamento

- In C l’assegnazione è un’espressione e dunque produce un risultato: questo risultato è il valore dell’espressione a destra del segno ‘=’
- L’assegnazione ha associatività da destra a sinistra, quindi è possibile assegnare lo stesso valore a più variabili con la scrittura seguente:

  ```
  a = b = c = 2;
  ```

  che equivale a:

  ```
  a = (b = (c = 2));
  ```

# Operatori di assegnamento

- La forma di assegnamento:
  *variabile op***=** *espressione*
  dove *op* è un operatore del C, equivale a:
  *variabile* **=** *variabile op espressione*
  `x += 5;`
  equivale a: `x = x + 5;`
- Esiste per tutti gli operatori aritmetici e bitwise:
  `+= -= *= /= %= &= ^= |= <<= >>=`
- `vett[y%(x+2)] += 5;`
  questo esempio mostra il vantaggio di non dover scrivere due volte la quantità a sinistra, inoltre questa viene valutata una volta sola, si comprende dopo aver studiato i vettori

# Funzioni matematiche

- Sono contenute in una libreria esterna al compilatore, collegata all'eseguibile dal linker
- Richiedono che venga incluso il file `<math.h>` che ne descrive la sintassi (mediante i *prototipi*, concetto descritto in altre slide)
- In genere elaborano valori `double` (indicati tra parentesi) e producono risultati `double`
- Se i valori passati non sono di tipo `double`, sono implicitamente convertiti in `double` in quanto corrispondono a veri e propri assegnamenti (per effetto dei prototipi)
- Le funzioni trigonometriche usano i radianti ($180^{\circ} = \pi$ rad)

# Funzioni matematiche

- `sin(x)` → seno
- `cos(x)` → coseno
- `tan(x)` → tangente
- `asin(x)` → arcoseno
- `acos(x)` → arcocoseno
- `atan(x)` → arcotangente → $[-\pi/2, +\pi/2]$
- `atan2(y,x)` → arcotangente di `y/x` → $[-\pi, +\pi]$ a seconda del valore di `x` e `y` (sia `x` sia `y` possono valere ±0)
- `exp(x)` → esponenziale $e^x$
- `log(x)` → logaritmo naturale
- `log10(x)` → logaritmo in base 10

# Funzioni matematiche

- `pow(x,y)` → $x^y$ (se `y`<0, `x` deve avere valore intero)
- `sqrt(x)` → radice quadrata (`x` ≥ 0)
- `fabs(x)` → valore assoluto
- `ceil(x)` → minimo intero ≥ `x`
- `floor(x)` → massimo intero ≤ `x`

- Esempi

```
y = sin(x*3.14/180);
y = sin(x*2)*2;
z = sqrt(fabs(x)*fabs(y));
z = log(x)/log(2);
Pi = 4.0*atan(1.0);
```

# Funzioni matematiche

- Alcune altre funzioni matematiche sono descritte nell'header file `<stdlib.h>`:

`abs(x)` → calcola il valore assoluto di un valore `int` e produce un risultato di tipo `int`

`labs(x)` → calcola il valore assoluto di un valore `long` e produce un risultato di tipo `long`

# Funzioni matematiche

- Per arrotondare un numero all'intero più vicino la libreria standard C89 non dispone di alcuna funzione
- Una soluzione statisticamente corretta richiede che i valori $x$.5 siano approssimati al valore pari più vicino, quindi:
  - **se $x$ è dispari:** per eccesso (in valore assoluto)
    +1.5 → +2 –1.5 → –2
  - **se $x$ è pari:** per difetto (in valore assoluto)
    +2.5 → +2 –2.5 → –2

# Funzioni matematiche

- Una soluzione semplice (non statisticamente corretta) è la seguente (richiede `float.h`):
  - Approssimazione sempre per eccesso:
    - per valori positivi:
      `(int)(valore+0.5)`
    - per valori negativi:
      `(int)(valore-0.5)`
  - Approssimazione sempre per difetto:
    - per valori positivi:
      `(int)(valore+(0.5-DBL_EPSILON))`
    - per valori negativi:
      `(int)(valore-(0.5-DBL_EPSILON))`

# Valori casuali

- Ad ogni chiamata, la funzione `rand` produce un diverso valore intero compreso tra 0 e `RAND_MAX` (estremi inclusi) con distribuzione uniforme

```
x=rand();
```

- `RAND_MAX` vale *almeno* 32767 (il valore effettivo dipende dal compilatore)
- Le funzioni e le definizioni delle costanti simboliche sono contenute in `<stdlib.h>`
- Per avere un valore intero tra 0 e N (escluso) il modo più semplice è il seguente:

```
x = rand() % N;
```

# Valori casuali

- I valori prodotti non sono realmente casuali, ma *pseudo-casuali:* la sequenza di valori prodotti è sempre la stessa e si ripete con un periodo molto lungo
- L'inizializzazione del generatore permette a `rand` di fornire valori dalla sequenza non partendo dal primo
- Per inizializzare il generatore bisogna includere `<time.h>` e scrivere (una sola volta) all'inizio del programma:
  `srand(time(NULL));`

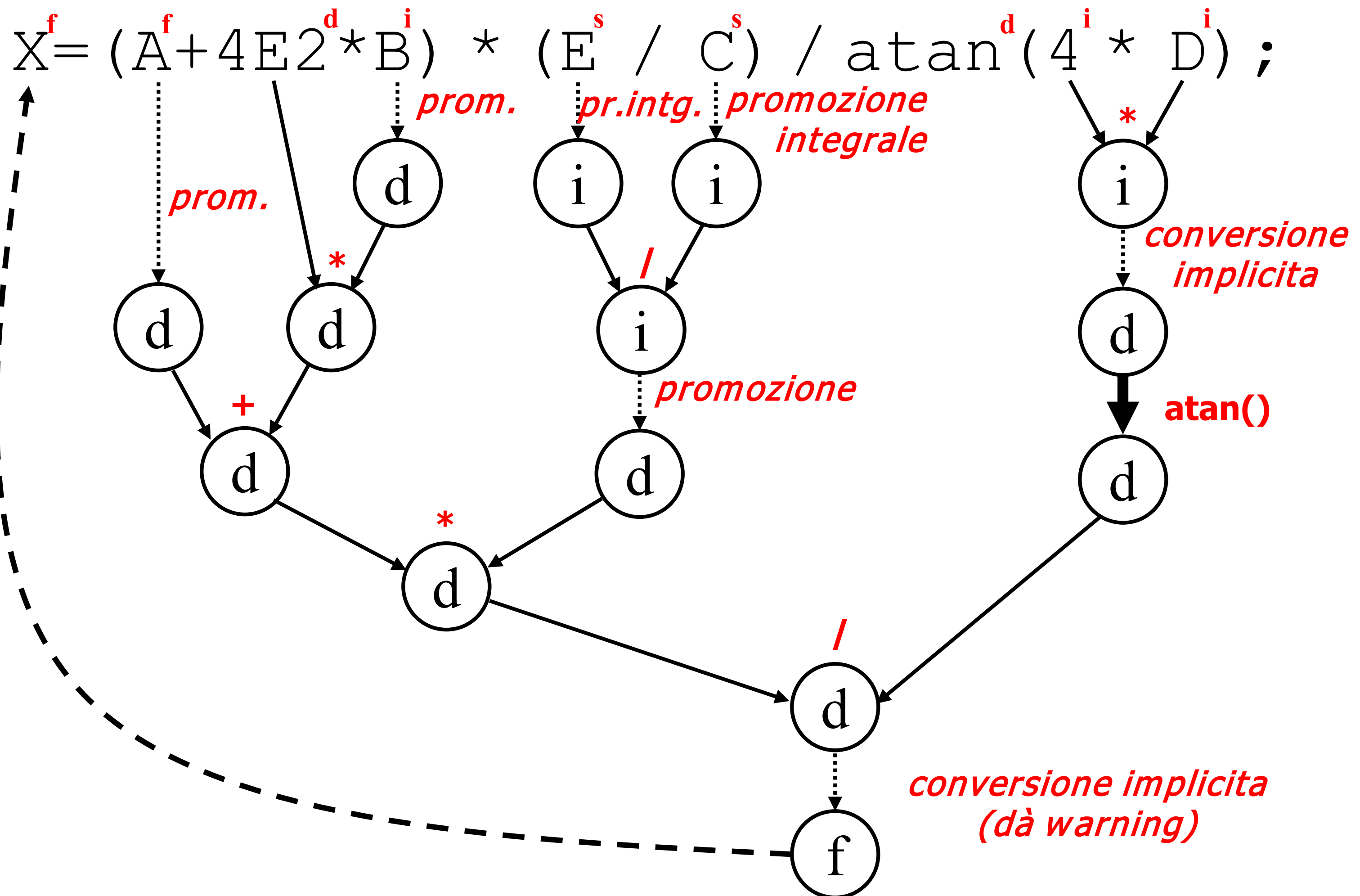
# Esercizi

0. Disegnare il diagramma di valutazione della seguente espressione, considerando le variabili definite come segue:

```
float A,X;
int B,D;
long C;
X=(A+4E2*B) * (C / 2) / atan(4.0F * D);
```

# Esercizi

0. Soluzione (nei cerchi il tipo prodotto)



# Esercizi

1. Scrivere un programma che chieda 4 numeri `int`, ne calcoli la media, la memorizzi in una variabile `float` e la visualizzi con 2 decimali.
2. Scrivere un programma che chieda un valore `double` di temperatura in gradi Fahrenheit e calcoli i valori delle corrispondenti temperature in gradi Celsius e Kelvin (entrambi con parte frazionaria).

$$C = \frac{5}{9} \cdot (F - 32)$$

$$K = C + 273.15$$

# Esercizi

3. Un oggetto che si muove ad una velocità $v$ confrontabile con quella della luce $c$ (2.99793·10$^8$ m/s) apparentemente si accorcia (nel senso della direzione) e aumenta di massa. Le due grandezze sono modificate da un fattore $\gamma$ (minore di 1) pari a:

$$\gamma = \sqrt{1 - \left(\frac{v}{c}\right)^2}$$

Si scriva un programma che chieda la lunghezza e la massa di un oggetto fermo e calcoli la variazione delle due grandezze a quella velocità (richiesta in input in km/sec).

# Esercizi

4. Si scriva un programma per calcolare la distanza in linea d'aria tra due punti della superficie terrestre, note le coordinate geografiche. Il programma chiede i valori di latitudine (N-S) e di longitudine (E-O) in gradi dei due punti. Per calcolare la distanza si usi la seguente formula (le coordinate Nord e Est sono positive, Sud e Ovest negative). Si ricordi che le funzioni trigonometriche utilizzano i radianti.

$$distanza = \arccos(\, p1 + p2 + p3 \,) \cdot r$$

# Esercizi

(*Continuazione*)

*dove:*

$r$ è il raggio medio della Terra: 6372.795 km

$p1 = \cos(lat1) \cdot \cos(lon1) \cdot \cos(lat2) \cdot \cos(lon2)$

$p2 = \cos(lat1) \cdot \sin(lon1) \cdot \cos(lat2) \cdot \sin(lon2)$

$p3 = \sin(lat1) \cdot \sin(lat2)$

essendo:

- $lat1$ latitudine in gradi del primo punto
- $lon1$ longitudine in gradi del primo punto
- $lat2$ latitudine in gradi del secondo punto
- $lon2$ longitudine in gradi del secondo punto

N.B. La formula considera la terra sferica con raggio medio $r$: non essendolo, la formula dà un risultato con un errore massimo dello 0.5%.

# Esercizi

(*Continuazione*)

Per la formula del arco-coseno non si utilizzi la funzione di libreria `acos`, ma la seguente:

$$\arccos(x) = \arctan\left(\frac{-x}{\sqrt{1-x^2}}\right) + \frac{\pi}{2}$$

Calcolare le distanze tra gli aeroporti di:

- Torino (TRN, 45.02° N, 07.65° E)
- Roma (FCO, 41.81° N, 12.25° E)
- Los Angeles (LAX, 33.94° N, 118.40° W)

Nota: $\pi = 4 \cdot \tan^{-1}(1)$

[Risposte: TRN-FCO: 515.20 km, TRN-LAX: 9692.7 km, FCO-LAX: 10205.48 km (W implica un valore <0)]