Costanti

Costanti intere

- Vi sono 3 rappresentazioni:
  - in base 10, se scritte come interi che non iniziano per `0`;
  - in base 8, se scritte come interi che iniziano per `0`;
  - in base 16, se scritte come interi che iniziano per `0x` o `0X`.
    - Le eventuali lettere possono essere sia maiuscole che minuscole.
- Attenzione: le costanti ottali o esadecimali non sono **mai negative**.

Tipo delle costanti intere

- Il loro tipo è il **minimo tipo intero**, iniziando per default da `int`, che può rappresentarne il valore.
  - Attenzione quindi: il tipo **dipende dal valore e dall'implementazione**!
- Possono essere seguite da un suffisso per modificarne il tipo:
  - se il suffisso è `L` o `l`, si inizia da `long`;
  - se il suffisso è `LL` o `ll`, si inizia da `long long`;
  - se il suffisso inizia o termina con `U` o `u`, il tipo diventa `unsigned`.

Esempi di costanti intere

```
23   // Costante decimale
023  // Costante ottale: vale 19
0x23 // Costante esadecimale: vale 35
0xA  // Costante esadecimale: vale 10
1L   // Costante di tipo long
1LL  // Costante di tipo long long
9u   // Costante di tipo unsigned int
9ul  // Costante di tipo unsigned long
9l   // Costante di tipo long
```

Usate suffissi **maiuscoli**: l'ultima costante vale 9, non 91!

Costanti carattere

- Si rappresentano con un carattere tra apici.
- Sono in realtà di tipo `int`!
- Per rappresentare caratteri non stampabili o non disponibili sulla tastiera nelle costanti carattere o stringa si possono usare le **escape sequence**:
  - escape sequence ottale: `\` seguito da fino a 3 cifre ottali;
  - escape sequence esadecimale: `\x` seguito da cifre esadecimali;
  - caratteri universali;
  - escape sequence semplici.
    - Il modo normale e portabile per scrivere <return> <tab> e caratteri simili.

Escape sequence semplici

Sequenza	Carattere equivalente
`\a`	alert (un suono o un carattere convenzionale)
`\b`	backspace (porta indietro la posizione di stampa)
`\f`	form feed (salto pagina)
`\n`	newline (a capo)
`\r`	carriage return (ritorno a inizio riga)
`\t`	horizontal tab (tabulazione orizzontale)
`\v`	vertical tab (tabulazione verticale)
`\?`	`?` (punto interrogativo)
`\'`	`'` (apice)
`\"`	`"` (doppio apice)
`\\`	`\` (backslash)

Esempi di costante carattere

```
char c = 'a';

if (c >= '0')
  …
```

Conversione da numero (tra 0 e 9) a cifra

```
c = '0' + n;
```

Esempi di escape sequence

```
char c = '\n';
```

Assegna il terminatore di stringa

```
c = '\0';
```

Fai “beep” e vai a capo

```
printf("\a\n");
```

Cancella l’ultimo carattere scritto e arretra il cursore (notare lo spazio)

```
printf("\b \b");
```

Costanti di tipo wchar_t

- Si rappresentano come le costanti carattere, premettendo una `L` (obbligatoriamente maiuscola) agli apici.
  - Per esempio: `L'A'`, `L'\n'`, `L'\x0461'`
- Si rappresentano frequentemente con escape sequence, perché i caratteri corrispondenti non sono gestibili facilmente.

Costanti booleane

- Sono `true` e `false`.
- Non sono predefinite, ma macro definite in stdbool.h.

Costanti reali (1)

- Vi sono 3 rappresentazioni:
  - in base 10, con notazione consueta;
  - in base 10, con notazione esponenziale, se è presente l'esponente;
  - in base 16, con notazione esponenziale, se iniziano con `0X` o `0x`.
- Una costante è reale se contiene il punto decimale o l'esponente o un suffisso che indica un tipo reale.
  - Attenzione quindi: `01e1` è reale!

Costanti reali (2)

- L’esponente:
  - per la notazione decimale è indicato dalle lettere `E` o `e` e la base è 10;
  - per la notazione esadecimale è indicato dalle lettere, `P` o `p` e la base 2.
- Il punto decimale deve essere preceduto o seguito da almeno una cifra.
  - Un punto isolato non è una costante reale valida.
- L’eventuale segno fa parte della costante.

Tipo delle costanti reali

- Per default sono di tipo `double`.
- Se seguite dal suffisso `F` o `f` sono di tipo `float`.
- Se seguite dal suffisso `L` o `l` sono di tipo `long double`.

Esempi di costanti reali

```
123.4    // tipo double
1.
.1
1.0

.125E5   // equivale a 12500.0
.125E-4  // equivale a 0.0000125

0.1F     // tipo float
1.591L   // tipo long double
```

Operatori ed espressioni

Espressioni

- Sono formate da **operandi** (espressioni, variabili e costanti) e **operatori** (simboli o parole chiave che indicano un’operazione), eventualmente racchiusi tra parentesi.
- Hanno sempre un tipo, determinabile in compilazione.

Operatori

Categoria	Simboli
Aritmetici	`+ - * / %`
Incremento e decremento	`++ --`
Agenti sui bit	`~ & | ^ << >>`
Relazionali	`== != > >= <=`
Logici	`! && ||`
Conversione (cast)	`(<tipo>)`
Condizionale	`? :`
Virgola	`,`
Sizeof	`sizeof`
Assegnamenti	`= op=`
Accesso a vettori e record	`* & [] . ->`
Chiamata di funzione	`()`

Esempi di espressioni

```
x
5
a + b * c
sqrt(2.0)
(a + b) * fun(c - 5)
```

Promozione a intero (1)

- Quando un valore di un tipo intero più piccolo di `int` (cioè `_Bool`, `char` e `short int`) è usato in un'espressione, viene convertito a `int`, secondo le seguenti regole:
  - i valori di tipo `_Bool` vengono convertiti a `unsigned int`;
  - i valori di tipo `signed short int` e `signed char` vengono convertiti a `signed int`;
  - i valori di tipo `unsigned short int` e `unsigned char` vengono convertiti a `signed int`, se tale tipo può rappresentare tutti i valori del tipo originale, a `unsigned int` altrimenti.

Promozione a intero (2)

- **Attenzione**: la promozione a intero di un `unsigned` può quindi produrre un tipo `signed`.
  - Quando si usano valori `unsigned char` o `unsigned short` in un'espressione, conviene convertirli **esplicitamente** al tipo desiderato (con l'operatore cast), per evitare dipendenze dall'implementazione.
- Nelle espressioni, **per prima cosa** viene effettuata la promozione a intero di **tutti** gli operandi (tranne per gli assegnamenti).
  - Quindi anche semplici operazioni come il cambio di segno cambiano il tipo di un `char` o `short`.

Conversioni automatiche (1)

- Se gli operandi di un'operazione binaria, esclusi shift e assegnamenti, dopo la promozione a intero sono di tipi differenti, vengono convertiti entrambi al tipo maggiore secondo la seguente gerarchia:
  - `long double`,
  - `double`,
  - `float`,
  - `long long int`,
  - `long int`,
  - `int`.

Conversioni automatiche (2)

- Se gli operandi interi di un'operazione binaria sono uno di un tipo `signed` e l'altro di un tipo `unsigned`, il tipo al quale vengono convertiti è:
  - il tipo `unsigned`, se è superiore nella gerarchia;
  - il tipo `signed`, se può rappresentare tutti i valori dell'altro;
  - il tipo `unsigned` corrispondente al tipo `signed`, altrimenti.

Conversioni automatiche (3)

- Se uno degli operandi di un'operazione binaria è di un tipo complesso, anche l'altro viene convertito a complesso.
- Il tipo finale viene determinato poi con la solita gerarchia.
  - Per esempio, una somma tra un `double` e un `float _Complex` produce un risultato di tipo `double _Complex`.

Esempi di conversioni automatiche

```
signed char    c1, c2;
int            i;
long           l;
float          f;

f + (i * l)
```

equivale a:

```
f + (float)((long)i * l)
```

```
c1 + c2
```

equivale a:

```
(int)c1 + (int)c2
```

Pericoli nelle conversioni (1)

- Dalle regole segue che il tipo di alcune espressioni coinvolgenti interi `signed` e `unsigned` dipende dall'implementazione.
- Quindi bisogna utilizzare l'operatore cast per assicurare la portabilità nei seguenti casi:
  - quando un operando è `unsigned char` o `unsigned short` e l'altro è `signed`;
  - quando un operando è `_Bool` o `unsigned` e l'altro è `signed`.

Operatori aritmetici

- Ci sono 2 operatori aritmetici unari:

  `+`, `-`, con operando di qualsiasi tipo aritmetico.

- Ci sono 5 operatori aritmetici binari:

  `+`, `-`, `*`, `/`, tra operandi di qualsiasi tipo aritmetico.

  `%` per il calcolo del resto, solo con operandi interi.

Operatori agenti sul segno

- Indicati dai simboli + e -.
  - L'operando deve essere di un tipo aritmetico;
  - Il valore prodotto è del tipo dell'operando.
- Il valore dell'espressione è:
  - per l'operatore +, il valore dell'operando inalterato;
  - per l'operatore -, il valore dell'operando col segno cambiato.

Addizione e sottrazione

- L’addizione è indicata dall’operatore `+`.
  - E’ commutativa e associativa tra interi, se non si verificano overflow.

    `a + b` = `b + a`.

    `a + b + c` = `(a + b) + c` = `a + (b + c)`.
  - E’ commutativa, ma **non associativa**, tra reali, a causa degli arrotondamenti.
- La sottrazione è indicata dall’operatore `-`.

  `a - b` equivale a `a + (-b)`.

Moltiplicazione

- Indicata dall'operatore `*`.
- E' commutativa e associativa tra interi, se non si verificano overflow.

  `a * b = b * a`.

  `a * b * c = (a * b) * c = a * (b * c)`.
- E' commutativa, ma **non associativa**, tra reali, a causa degli arrotondamenti.

Divisione

- La divisione è indicata dall'operatore `/`.
  - Tra interi, il quoziente viene **troncato verso zero**.
    - Quindi 5 / 2 dà 2 e -5 / 2 dà -2.
  - Tra reali, il quoziente viene **approssimato**.
    - Quindi 5.0 / 2.0 dà 2.5 e –5.0 / 2.0 dà –2.5.
- Il comportamento in caso di divisione per zero è **indefinito**.
  - Di solito l'esecuzione termina in errore.

Calcolo del resto

- Il calcolo del resto è indicato dall'operatore %.
  - Gli operandi devono essere interi.
- Il resto è definito come segue:

  `a % b = a - a / b * b`.
  - Il resto ha sempre lo stesso segno del dividendo.
- Il comportamento in caso di modulo con secondo operando uguale a zero è **indefinito**.
  - Di solito l'esecuzione termina in errore.

lvalue

- Termine usato per indicare **un'entità modificabile**, che puo essere
  - una variabile;
  - un campo di record o unione;
  - un elemento di vettore;
  - un espressione di tipo puntatore, preceduta dall'operatore *.

Operatori di incremento e decremento (1)

- Indicati dai simboli `++` e `--`.
- Incrementano (`++`) o decrementano (`--`) l'operando.
  - L'operando deve essere di un tipo aritmetico o puntatore.
  - L'operando deve essere un lvalue, non può essere un'espressione qualsiasi.

Operatori di incremento e decremento (2)

- Il valore prodotto è del tipo dell'operando, senza alcuna conversione.
- Il valore dell'espressione è:
  - nella versione prefissa (es.: `++n`), il valore della variabile **dopo** l'incremento o il decremento;
  - nella versione postfissa (es.: `n--`), il valore **originale** della variabile.

Esempi di incremento e decremento (1)

- Se n vale 1, eseguendo

```
x = ++n;
```

oppure

```
x = n++;
```

il valore finale di n diviene 2, ma nel primo caso si assegna 2 a x, nel secondo si assegna 1.

Esempi di incremento e decremento (2)

```
int i = 5;
double d = 0.0;

j = i++;      // j vale 5, i diventa 6
k = ++i;      // i diventa 7, k vale 7
l = --i;      // i diventa 6, l vale 6
++d;          // d diventa 1.0
--i;          // i diventa 5
```

Errori aritmetici

- In caso di errore (overflow, underflow o divisione per zero) in un'operazione aritmetica il **comportamento** è indefinito.
  - Di solito si ha:
    - un segnale di errore (SIGFPE) nel caso di divisione per zero o di overflow tra reali;
    - zero in caso di underflow tra reali;
    - un risultato indefinito per le altre operazioni.

Aritmetica unsigned

- L'aritmetica dei tipi `unsigned` è sempre modulare, quindi non può mai verificarsi un overflow, neppure se il risultato di un'operazione è negativo.
  - Il valore di un'espressione `unsigned` viene calcolato come:

    <valore prodotto> = <valore vero> % (MAX + 1)
  - dove MAX rappresenta il massimo rappresentabile dal tipo, cioè `U`<*tipo*>`_MAX`.

Aritmetica reale (1)

- È soggetta a errori di arrotondamento: non bisogna **mai** aspettarsi risultati **esatti**.
  - Praticamente l'unica certezza che si ha è che i risultati sono **sbagliati**!
- Il confronto per uguaglianza di numeri reali o complessi **non è significativo**, a causa degli errori di arrotondamento.

  Per esempio:

  ```
  1.7 + 0.3 == 2.0
  ```

  può dare valore falso!

Aritmetica reale (2)

- I risultati sono deterministici: ripetendo i calcoli con gli stessi valori si ottengono gli stessi risultati.
  - Ma solo su macchine uguali o compatibili.
- Singole parti di espressioni possono però essere “contratte”, cioè calcolate con precisione **maggiore** di quella specificata, **riducendo** gli errori di arrotondamento rispetto a quelli che si verificherebbero se le operazioni venissero eseguite una per volta con la precisione richiesta.
- Tale contrazione può essere vietata per mezzo del pragma `FP_CONTRACT`.

Aritmetica reale (3)

- Di conseguenza espressioni uguali in istruzioni distinte possono dare valori diversi, pur operando sugli stessi valori.

  Per esempio, se le variabili sono tutte reali:

```
a = b * c – d * e;
x = b * c;
y = d * e;

a == x - y
```

  Il confronto finale può dare valore falso!

Operatori agenti sui bit

- In C vi sono 6 operatori definiti a partire dalla rappresentazione binaria degli operandi, invece che dal loro valore.
- Gli operandi devono essere di un tipo intero.

Operatore unario agente sui bit

- ~: inverte i bit dell'operando.
  - Se la rappresentazione è in complemento a 2, `~x` è uguale a `(-x) - 1`
- Esempi:

  ~5 = ~00...00101$_2$ = 11…11010$_2$ = -6;

  ~3 = 00...0011$_2$ = 11...1100$_2$.

  ~-9 = ~11…110111$_2$ = 00…001000$_2$ = 8.

  L'ultimo esempio vale solo se la rappresentazione è in complemento a 2.

Operatori binari agenti sui bit

- Il valore prodotto è dello stesso tipo degli operandi.

  &: and bit a bit degli operandi;

  |: or bit a bit degli operandi;

  ^: or esclusivo bit a bit degli operandi;

- Esempi:

  $5 \,\&\, 3 = 0...0101_2 \,\&\, 0...0011_2 = 0...0001_2 = 1;$

  $5 \mid 3 = 0...0101_2 \mid 0...0011_2 = 0...0111_2 = 7;$

  $5 \,\hat{}\, 3 = 0...0101_2 \,\hat{}\, 0...0011_2 = 0...0110_2 = 6.$

Shift (1)

- Provocano lo scorrimento del primo operando, del numero di bit indicato dal secondo:

  <<: shift a sinistra;

  >>: shift a destra.
- Il valore prodotto è del tipo del primo operando.
- Il risultato è definito **solo se** il numero di shift è **non negativo** e **minore** del numero di bit del primo operando.

Shift (2)

- I bit che entrano sono:
  - zeri per gli shift a sinistra e per gli shift a destra di valori non negativi;
  - **indefiniti** per shift a destra di valori negativi.
- I bit che escono sono persi.
- Di conseguenza andrebbero utilizzati solo con operandi `unsigned`.
  - Come regola, **tutti** gli operatori agenti sui bit dovrebbero avere operandi `unsigned`.

Esempi di shift

- Esempi:

  `5 << 3` = $0...0101_2$ << 3 = $0...0101000_2$ = 40;

  `5 >> 2` = $0...0101_2$ >> 2 = $0...0001_2$ = 1;

  `-9 >> 2` = $1\ldots110111_2$ >> 2 = $0011\ldots1101_2$ = 2^{30} - 3;

Caratteristiche degli operatori agenti sui bit

- Dato che per valori non negativi il legame tra valore e rappresentazione è fisso, il risultato delle operazioni può anche essere definito in termini aritmetici.
  - Comunque chi li usa al posto degli operatori aritmetici (per esempio shift al posto di moltiplicazioni) è un analfabeta.

Operatori relazionali (1)

- Il C dispone dei 6 consueti operatori di confronto:
  - `>` per indicare maggiore;
  - `>=` per indicare maggiore o uguale;
  - `<` per indicare minore;
  - `<=` per indicare minore o uguale;
  - `==` per indicare uguale;
  - `!=` per indicare diverso.
- Gli operandi possono essere numerici o puntatori.
  - Con i numeri complessi sono validi solo i confronti `==` e `!=`.

Verità e falsità

- Quando un'espressione viene esaminata per decidere se è vera o falsa, si considera **falso** il valore **zero** per i tipi aritmetici e il **puntatore nullo** per i puntatori; **qualsiasi altro** valore è considerato **vero**.
  - Meglio comunque evitare di scrivere:

  `if (n)` o `if (p)`

  è **nettamente preferibile**:

  `if (n != 0)` o `if (p != NULL)`
- Quando un operatore produce un valore di verità, produce un `int`, che vale 0 per falso e 1 per vero.
  - Pertanto sono ammesse espressioni orribili come:

```
n = (a > 0) * 5
```

Operatori logici

- Uno è unario:

  `!`, che indica not logico (inverte il valore di verità dell'operando).
- Due sono binari:

  `&&`, che indica and logico;

  `||`, che indica or logico.
- Producono un valore di tipo `int`.

Valutazione abbreviata

- Gli operatori && e || valutano **sempre** il primo operando, il secondo **solo se necessario.**
  - Ossia se il primo è falso per && e se è vero per ||.
- Questo permette espressioni pericolosamente delicate e oscure:

```
if (n >= 0 && v [n] == 0)
…
```

Valida solo perché se n è negativo non viene usato come indice e l'esame di n ne precede l'uso.

Esempio di operatori relazionali e logici (1)

- Funzione che determina se un anno è bisestile.

```
int leap(int year)
    {
    if ((year % 4 == 0 &&
         year % 100 != 0) ||
         year % 400 == 0)
         return 1; // leap
    else
         return 0; // not leap
    }
```

Esempio di operatori relazionali e logici (2)

- Versione più corretta come dichiarazione di tipi.

```
bool leap(unsigned int year)
   {
   return (year % 4 == 0 &&
   year % 100 != 0) ||
   year % 400 == 0;
   }
```

Attenzione alle confusioni!

- Scambiando & con && o | con || si ottengono generalmente espressioni legali, ma il valore prodotto cambia!
  - Per esempio, se `a` vale 1 e `b` vale 2:

    `a & b` è 0, ma `a && b` è 1;

    `a | b` è 3, `ma a || b` è 1.

Operatore di conversione

- L’operatore unario di conversione (detto cast) si scrive mettendo il tipo al quale convertire l’espressione tra parentesi, prefisso all’espressione.
  - Es.: `(long)(x + y)`.
- Il tipo deve essere aritmetico o puntatore.

Conversioni tra tipi interi

- Nelle conversioni, implicite o esplicite, tra tipi interi o da tipo intero a reale, se il valore può essere rappresentato nei limiti del tipo al quale si converte, resta inalterato.
- Altrimenti:
  - Se il tipo al quale si converte è `unsigned`, il risultato viene calcolato modulo il massimo rappresentabile (cioè `U`<*tipo*>`_MAX`) più uno.
  - Se il tipo verso il quale si converte è `signed`, il **risultato** dipende dall'implementazione.

Conversioni da tipi reali a tipi interi

- Nelle conversioni, implicite o esplicite da tipi reali a tipi interi si preserva il valore, se possibile, **troncando verso zero** (scartando la parte frazionaria).
- Se valore non può essere rappresentato nei limiti del tipo, il **comportamento** dipende dall'implementazione.
- Attenzione: nelle conversioni da reale a intero `unsigned` non è garantito che si applichi il modulo per valori maggiori di `U<`*tipo*`>_MAX`.

Uso corretto dell’operatore cast

- Gli unici utilizzi ammessi dell'operatore cast tra tipi aritmetici sono:
  - forzare conversioni in assegnamenti;
  - forzare conversioni nel passaggio di argomenti a funzioni;
  - forzare conversioni nella produzione di valori di funzione;
  - controllare precisione di espressioni;
  - controllare l'intervallo di rappresentazione di espressioni;
  - cast di tipi enumerati a interi, per utilizzarli come indici;
  - cast di somme di tipi enumerati e interi al tipo enumerato di partenza;

Esempi di cast (1)

```
int      j = 3;
int      k = 2;
float    f;
```

`(long)1` equivale a `1L`

```
f = j / k;                    // f = 1.0
f = (float)j / (float)k;      // f = 1.5
f = (float)j / k;             // f = 1.5
```

Esempi di cast (2)

```
long      l;
int       i;
float     f;
```

Conversione in assegnamento:

```
  i = (int)l;
```

Passaggio di argomenti:

```
  f = (float)sin((double)i);
```

Valore di funzione:

```
  return (float)3.5;
```

Esempi di cast (3)

Se `i`, `j` e `k` sono `int`, le espressioni:

`k = (i + j) / 2` o `k = i * j / 1000`

possono dare risultati errati causa overflow.

Possibili soluzioni:

```
k = (int)(((long)i + (long)j) / (long)2)
k = ((long)i + j) / 2    // meno chiaro
k = (i * 1L + j) / 2     // atroce!

k = (int)(((long)i * (long)j) / (long)1000)
k = ((long)i * j) / 1000 // meno chiaro
k = i * 1L * j / 1000    // atroce!
```

Esempi di cast (3)

```
float   f, f1, f2, f3;
```

Controllo precisione:

```
f = (float)(((double)f1 * f2) - f3);
```

Meglio:

```
f = (float)(((double)f1 * (double)f2)-
  (double)f3);
```

Esempi di cast di enumerati

```
enum month_type
  {
  JAN = 0, FEB, MAR, APR, MAY, JUN,
  JUL, AUG, SEP, OCT, NOV, DEC
  } month;

month = JAN;
printf("%ld\n",
  payroll [(unsigned int)month]);
month = (enum month_type)
  ((unsigned int)month + 6);
```

Riassunto sulle conversioni

Da tipo	A tipo	Il valore viene
intero	intero	conservato
reale	intero	troncato
qualsiasi	reale	approssimato

Operatore condizionale

- Operatore ternario, indicato dai simboli `?` e `:`.
  - Valuta il primo operando e produce come valore il secondo, se il primo è vero, il terzo se è falso.
  - **Uno solo** tra secondo e terzo operando viene valutato, ma non entrambi.
- Equivale a un if - else.
- Associativo **da destra**:

  `a ? b : c ? d : e ? f : g` equivale a

  `a ? b : (c ? d : (e ? f : g))`

Esempi di operatore condizionale (1)

- Invece di

```
if (a > b)
   max = a;
else
   max = b;
```

si puo scrivere

```
max = a > b? a : b;
```

Esempi di operatore condizionale (2)

- Scrivi un vettore di n elementi, 10 per riga, con gli elementi su ogni riga separati da uno spazio:

```
for (i = 0; i < n; ++i)
  printf("%6d%c", a [i],
   (i % 10 == 9 || i == n - 1)?
   '\n': ' ');
```

Vincoli sull'operatore condizionale

- Secondo e terzo operando devono essere di tipi **compatibili**.
  - Se sono di tipi aritmetici, vengono convertiti allo stesso tipo, secondo le regole delle conversioni aritmetiche per operatori binari.
  - Se sono puntatori, devono essere di tipi compatibili.
  - Possono essere entrambi tipi strutturati (record, unioni), ma dello stesso tipo.
- Non produce un lvalue:

  `(a > b? a: b) = c;` è illegale.

Operatore virgola

- Valuta il primo operando, ignora il risultato, valuta il secondo.
- Valore e tipo sono quelli del secondo operando.
- Si usa principalmente nell'istruzione `for`.

Esempio di operatore virgola

- Inverti l'ordine degli elementi di un vettore di n elementi.

```
for (i = 0, j = n - 1; i < j; +
  +i, --j)
  {
  temp = v [i];
  v [i] = v [j];
  v [j] = temp;
  }
```

Attenzione all'operatore virgola!

- Usato per sbaglio al posto del punto, provoca errori che sfuggono al compilatore e a una lettura non attentissima.

```
pi = 3,14159265;   // pi vale 3

volume = r * r * r *
  (3,14159265 * 4 / 3);
```

Nei due casi la virgola è un operatore **legale**!

Operatore sizeof (1)

- Produce la dimensione in byte dell'operando.
  - Il byte è definito come la dimensione della memoria allocata per un `char`, quindi per definizione `sizeof(char)` = 1.
- Ha due sintassi:

  `sizeof(`<tipo>`)`

  `sizeof` <espressione>

  Le parentesi non sono necessarie nel secondo caso.

Operatore sizeof (2)

- Il tipo del valore prodotto è `size_t`, definito in stddef.h (incluso in stdio.h e stdlib.h) come il minimo tipo intero `unsigned`, (a partire da `unsigned int`) che permetta di rappresentare la dimensione di qualsiasi oggetto.
  - Generalmente è `unsigned int`.
  - E' di un tipo più grande, se `unsigned int` non basta a rappresentare il numero di elementi del più grande tipo gestibile.

Operatore di assegnamento semplice

- Si indica col simbolo =.
- L'operando sinistro deve essere un **lvalue**, non può essere un'espressione qualsiasi.
- Il valore viene **convertito** al tipo del primo operando.
  - Se la conversione è impossibile (i tipi non sono compatibili) si ha un errore.
- Produce come valore il valore assegnato.
- E' associativo da destra.

Esempi di assegnamento semplice

```
a = b + c;
a = b = c = 0;
a = (b = x + y) * (c = z * 2);
while ((c = getc(f)) != EOF)
   …
```

Tranne nel caso di assegnamenti multipli di un unico valore a variabili dello stesso tipo, è meglio **evitare** di scrivere più assegnamenti nella stessa istruzione.

Esempi di conversioni in assegnamento

```
signed char     c1, c2, c3;
int             i
double          d;

d = i;      equivale a:    d = (double)i;

c1 = c2 + c3;
```

equivale a:

```
c1 = (char)((int)c2 + (int)c3);

c1 = 3.9;  equivale a:    c1 = 3;
```

Attenzione all'assegnamento!

- Scambiando `=` con `==` si ottengono spesso espressioni legali, ma l'effetto cambia!
  - Per esempio:

  ```
  a == 0;
  ```

  Doveva essere `a = 0`; ma così `a` non viene modificata.

  ```
  if (a = 1)
  ```

  Doveva essere `if (a == 1)` ma così `a` viene modificata e l'espressione è sempre vera.

Operatori di assegnamento composti

- Ci sono 10 operatori di assegnamento composti, scritti come *op*=, dove *op* è un operatore binario, aritmetico o agente sui bit.
- Equivalgono a una combinazione di *op* e =:

  <var> *op*= <expr> equivale a <var> = <var> *op* <expr>.

  Sono possibili le seguenti combinazioni: `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `|=` `^=`
- Producono come valore il valore assegnato.
- L'operando sinistro deve essere un lvalue.

Esempi di assegnamenti composti

```
x += y;  // Equivale a x = x + y;
n += 1;  // Equivale a ++n;

x += y *= z;
  // Equivale a x = x + (y = y * z);
```

Per non rendere illeggibile il codice, è meglio evitare di scrivere più assegnamenti composti nella stessa istruzione.

Esempio di espressioni

- Funzione che conta i bit a 1 nell'argomento.

```
unsigned int bitcount(unsigned int x)
    {
    unsigned int   bits;

    bits = 0;
    for (; x != 0; x >>= 1)
         bits += (x & 0x1);
    return bits;
    }
```

Espressioni costanti

- In tutti i casi nei quali la sintassi richiede una costante, è ammessa un'espressione costante, cioè un'espressione che non contenga:
  - accesso a variabili;
  - chiamata di funzioni;
  - operatori unari `++`, `--`, `*` e `&`;
  - operatore `sizeof` con operando di dimensioni variabili;
  - operatori di assegnamento.

Precedenza e associatività (1)

- L’ordine di valutazione degli operatori è definito dalle regole di **precedenza** e **associatività**.
  - Vengono prima valutate le espressioni nelle parentesi piu interne.
  - A parità di livello di parentesi, gli operatori sono valutati in ordine di **precedenza**.
  - A parità di precedenza, gli operatori sono valutati in ordine di **associatività**.

Precedenza e associatività (2)

- L' associatività è sempre da **sinistra a destra**, tranne per i seguenti operatori:
  - tutti gli operatori unari;
  - tutti gli operatori di assegnamento;
  - l'operatore condizionale `?:`.
- Quando si desidera imporre un ordine di valutazione differente o si hanno dubbi sulla precedenza, basta usare le parentesi.

Precedenza degli operatori

Operatori	Associatività
`() [] -> .`	Da sinistra
`+ - * &` (tutti unari) `! ~ ++ --` cast `sizeof`	Da destra
`*` (binario) `/ %`	Da sinistra
`+ -` (binari)	Da sinistra
`<< >>`	Da sinistra
`< <= > >=`	Da sinistra
`== !=`	Da sinistra
`&`	Da sinistra
`^`	Da sinistra
`|`	Da sinistra
`&&`	Da sinistra
`||`	Da sinistra
`? :`	Da destra
`= += -= *= /= %= &= ^= |= <<= >>=`	Da destra
`,`	Da sinistra

Esempi di precedenza

- Nell'espressione `a * (b + c)` viene eseguita prima l'addizione (diverso livello di parentesi).
- Nell'espressione `a + b * c` viene eseguita prima la moltiplicazione (diversa precedenza).
- Nell'espressione `a * b / c` viene eseguita prima la moltiplicazione (associatività da sinistra).
- Nell'espressione `a - -b` viene eseguito prima il cambio di segno di `b` (diversa precedenza).

Ordine di valutazione degli operandi

- L'ordine di valutazione degli **operandi** è invece **indefinito**, tranne che per i quattro operatori &&, `||`, `?:` e `,`.
  - Bisogna quindi **evitare** espressioni che dipendano dall'ordine di valutazione degli operandi.
  - In particolare, l’assegnamento è un operatore, quindi non è garantito che l'indirizzo della variabile alla quale si assegna venga calcolato **prima** dell'espressione.

Esempi tipici di espressioni scorrette (1)

```
for (i = 0; i < 10; )
    v [i] = i++;
```

Non è definito se assegni `0` a `v [0]`, `1` a `v [1]` ecc. oppure `0` a `v [1]`, `1` a `v [2]` ecc..

```
i = 3;
a = ++i * (i + 1);
```

`a` può valere: 4 * (3 + 1) = 16 o 4 * (4 + 1) = 20.

Esempi tipici di espressioni scorrette (2)

```
a = f() + g();
```

Se `f` modifica una variabile usata da `g` o viceversa, il risultato è indefinito (`f` può essere chiamata sia **prima** che **dopo** `g`).

```
a = (f() + 5) * g(y);
```

Come sopra: le parentesi **non** forzano un ordine di valutazione degli **operandi**.

Regola generale

- Se un’espressione modifica il valore di una variabile, questa **non deve comparire** un’altra volta nella stessa espressione.

- Sfortunatamente esistono molti modi di modificare una variabile:
  - con un operatore di assegnamento;
  - con gli operatori di autoincremento e autodecremento;
  - chiamando una funzione che la modifica;
  - nei modi elencati, utilizzando puntatori.