14. Tipi e conversioni di tipo

Andrea Marongiu

(andrea.marongiu@unimore.it)

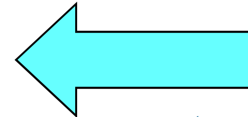
Paolo Valente

UNIMORE

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Tipi di dato primitivi

- **Enumerati (`enum`)**
- **Numeri reali (`float` e `double`)**
- **Tipi e conversioni di tipo**



  - Completamento dell'argomento aperto con le conversioni di tipo esplicite nella precedente lezione

Tipi primitivi 1/4

- Tipi interi — Dimensioni tipiche
  - **`int`** (32 bit)
  - **`short int`** (o solo **`short`**) (16 bit)
  - **`long int`** (o solo **`long`**) (64 bit)

- Tipi naturali
  - **`unsigned int`** (o solo **`unsigned`**) (32 bit)
  - **`unsigned short int`** (o solo **`unsigned short`**) (16 bit)
  - **`unsigned long int`** (o solo **`unsigned long`**) (64 bit)

- Un oggetto *unsigned* ha **solo valori maggiori o uguali di 0**

Tipi primitivi 2/4

- Per la precisione, il tipo **`long int`** è garantito avere almeno le stesse dimensioni del tipo **`int`**
- Siccome il tipo **`int`** è tipicamente su 32 bit, questo ha portato al problema che su molti compilatori il tipo **`long int`** è a 32 bit, mentre su altri è a 64 bit
- Per evitare tale problema, a partire dallo standard C++11, è disponibile anche il tipo **`long long int`**
  - E' garantito avere almeno le stesse dimensioni del tipo **`int`**, ma non meno di 64 bit

Tipi primitivi 3/4

- Tipo carattere — Dimensioni tipiche
  - **`char`** (8 bit)
  - **`signed char`** (8 bit)
  - **`unsigned char`** (8 bit)
  - Come già discusso, a seconda delle implementazioni **`char`** è implicitamente **`signed`** (può avere anche valori negativi) o **`unsigned`**

- Tipo reale
  - **`float`** (32 bit)
  - **`double`** (64 bit)
  - **`long double`** (80 bit)

Tipi primitivi 4/4

- Tipo booleano
  - **`bool`**

- Tipo enumerato
  - **`enum`** <*nome_tipo*> {<*lista_nomi_costanti*>}
  - A partire dallo standard C++11, anche
    - **`enum class`** <*nome_tipo*> {<*lista_nomi_costanti*>}

Compendio espressioni letterali

- Mediante i seguenti suffissi si possono scrivere espressioni letterali dei seguenti tipi:
  - U → unsigned int Es.: 3U
  - UL → unsigned long Es.: 3212UL
  - ULL → unsigned long long Es.: 1231ULL

Domanda

- Che succede se si decrementa di una unità una variabile di tipo **`unsigned int`** oppure **`unsigned char`** che contiene il valore 0?

Risposta

- Si ha un overflow !!!!
  - Per il momento diciamo che nella variabile finisce un valore casuale
  - Tale valore casuale potrebbe essere minore di 0?

Risposta

- No
  - Qualsiasi configurazione di bit utilizzata per rappresentare un numero senza segno rappresenta sempre un numero positivo o nullo

Precisazione unsigned

- In effetti, tecnicamente, con i tipi discreti e senza segno non c'è overflow
  - Perché le operazioni sono effettuate modulo il valore massimo per il tipo di dato
- Ad esempio, se MAX_UINT è una costante di tipo **`unsigned int`** contenente il valore massimo per il tipo **`unsigned int`**, allora
  - MAX_UINT + 1U = 0
  - 0 – 1 = MAX_UINT
  - MAX_UINT + 2U = 1
  - ...

Limiti 1/3

- In C++, includendo **`<limits>`** si possono utilizzare le seguenti espressioni:

numeric_limits<nome_tipo>::min()

valore minimo per il tipo nome_tipo

numeric_limits<nome_tipo>::max()

valore massimo per il tipo nome_tipo

numeric_limits<nome_tipo>::digits

numero di cifre in base 2

numeric_limits<nome_tipo>::digits10

numero di cifre in base 10

numeric_limits<nome_tipo>::is_signed

true se nome_tipo ammette valori negativi

numeric_limits<nome_tipo>::is_integer

true se nome_tipo e' discreto (int, char, bool, enum, ...)

Limiti 2/3

- Le seguenti informazioni hanno significato per i numeri in virgola mobile:

numeric_limits<nome_tipo>::epsilon()

minimo valore tale che 1 + epsilon != 1

numeric_limits<nome_tipo>::round_error()

errore di arrotondamento

numeric_limits<nome_tipo>::min_exponent

esponente minimo in base 2, cioè valore minimo esp, tale che il numero di possa scrivere nella forma m*(2^esp)

numeric_limits<nome_tipo>::min_exponent10

esponente minimo in base 10, cioè valore minimo esp, tale che il numero di possa scrivere nella forma m*(10^esp)

Limiti 3/3

... continua per i numeri in virgola mobile:

numeric_limits<nome_tipo>::max_exponent

esponente massimo in base 2, cioè valore massimo esp, tale che il numero di possa scrivere nella forma m*(2^esp)

numeric_limits<nome_tipo>::max_exponent10

esponente massimo in base 10, cioè valore massimo esp, tale che il numero di possa scrivere nella forma m*(10^esp)

- Esercizio: *limiti.cc* della settima esercitazione

Espressioni eterogenee

- Non ci sono dubbi sul comportamento di un operatore fin quando tutti i suoi operandi sono dello stesso tipo, ossia sono, come si suol dire, **omogenei**
- Ma cosa succede, per esempio, con l'operatore di assegnamento se un valore di un certo tipo viene assegnato ad una variabile di un tipo diverso?
- E cose succede con un qualsiasi altro operatore binario se viene invocato con due argomenti di tipo diverso?
- Nomenclatura: nei precedenti due casi siamo in presenza di operandi di tipo **eterogeneo**
- In generale, definiamo eterogenea una espressione che contenga fattori o termini di tipo eterogeneo

Conversioni di tipo

- In presenza di operandi eterogenei per un dato operatore si hanno due possibilità:
  - Il programmatore inserisce **conversioni esplicite** per rendere gli operandi omogenee
  - Il programmatore non inserisce conversioni esplicite
    - In questo caso
      - se possibile, il compilatore effettua delle **conversioni implicite** (**coercion**),
      - oppure segnala errori di incompatibilità di tipo e la compilazione fallisce

Coercion

- Il C/C++ è un linguaggio a *tipizzazione forte*
  - Ossia il compilatore controlla il tipo degli operandi di ogni operazione per evitare operazioni illegali per tali tipi di dato o perdite di informazione
- Le conversioni implicite di tipo che non provocano perdita sono effettuate dal compilatore senza dare alcuna segnalazione
- Tuttavia, le conversioni implicite che possono provocare perdita di informazioni **non sono illegali**
  - Vengono tipicamente segnalate mediante ***warning***
- In generale le conversioni implicite avvengono a tempo di compilazione in funzione di un ben preciso insieme di regole
  - Vediamo prima le regole in caso di operandi eterogenei per operatori diversi dall'assegnamento, poi quelle in caso di assegnamenti eterogenei

Operandi eterogenei 1/2

- Regole utilizzate in presenza di **operandi eterogenei per un operatore binario diverso dall'assegnamento**
  - Ogni operando di tipo `char` o `short` viene convertito in `int`
  - Se, dopo l'esecuzione del passo precedente, gli operandi sono ancora eterogenei, si converte l'operando di tipo inferiore al tipo dell'operando di tipo superiore. La gerarchia dei tipi è:

CHAR < INT < UNSIGNED INT < LONG INT < UNSIGNED LONG INT < FLOAT < DOUBLE < LONG DOUBLE

- Oppure, trascurando gli unsigned:

CHAR < INT < FLOAT < DOUBLE < LONG DOUBLE

Operandi eterogenei 2/2

- A questo punto i due operandi sono omogenei e viene invocata **l'operazione relativa all’operando di tipo più alto**
  - Anche il risultato sarà quindi dello stesso tipo dell'operando di tipo superiore

Esempi

```
int a, b, c; float x, y; double d;
```

`a*b+c` → espressione omogenea (int)

`a*x+c` → espressione eterogenea (float): prima a e poi c sono convertiti in float

`x*y+x` → espressione omogenea (float)

`x*y+5-d` → espressione eterogenea (double): 5 è convertito in float, poi il risultato di x*y+5 viene convertito in double

`a*d+5*b-x` → espressione eterogenea (double): a viene convertito in double, così come l'addendo (5*b) e la variabile x

Assegnamento eterogeneo

- L’espressione a destra dell’assegnamento viene valutata come descritto dalle regole per la valutazione di un’espressione omogenea o eterogenea viste finora
- Se il **tipo del risultato** di tale espressione è diverso da quello della variabile a sinistra dell'assegnamento, allora viene **convertito al tipo di tale variabile**
  - Se il tipo della variabile è gerarchicamente uguale o superiore al tipo del risultato dell'espressione, tale risultato viene convertito al tipo della variabile probabilmente senza perdita di informazione
  - Se il tipo della variabile è gerarchicamente inferiore al tipo del risultato dell'espressione, tale risultato viene convertito al tipo della variabile con alto rischio rischio di perdita di informazione
    - dovuto ad un numero inferiore di byte utilizzati per il tipo della variabile oppure, in generale, ad un diverso insieme di valori rappresentabili

Esempi 1/2

```
int   i = 4;      char  c = 'K';    double  d = 5.85;

i = c;       // conversione da char ad int
i = c+i;     // conversione da char ad int di c
             // per il calcolo di (c+i)
             // e poi assegnamento omogeneo
d = c;       // char →  double d==75.
i = d;       // sicuro troncamento della parte decimale
c = d / i;   // evidente perdita di informazione
```

Esempi 2/2

`int i=6, b=5;      float f=4.;         double d=10.5;`

`d = i;` → assegnamento eterogeneo (double ← int) → 6.
(Converte il valore di i in double e lo assegna a d)

`i=d;` → assegnamento eterogeneo (int ← double) → 10
(Tronca d alla parte intera ed effettua l'assegnamento ad i)

`i=i/b;` → assegnamento omogeneo (int ← int) → 1

`f=b/f;` → assegnamento omogeneo (float ← float) → 1.25
(Converte il b in float prima di dividere, perché f è float)

`i=b/f;` → assegnamento eterogeneo (int ← float) → 1
(L'espressione a destra diventa float perché b è float, tuttavia quando si effettua l'assegnamento, si guarda al tipo della variabile i)

Esercizio

```
int a, b=2;          float x=5.8, y=3.2;

a = static_cast<int>(x) % static_cast<int>(y); // a == ?
a = static_cast<int>(sqrt(49)); // a == ?

a = b + x;                  // è equivalente a quale nota-
                            // zione con conversioni
                            // esplicite: ?

y = b + x;                  // è equivalente a: ?

a = b + static_cast<int>(x+y);      // a == ?

a = b + static_cast<int>(x) + static_cast<int>(y);
                            // a == ?
```

Soluzione

```
int a, b=2;        float x=5.8, y=3.2;

a = static_cast<int>(x) % static_cast<int>(y); // a == 2
a = static_cast<int>(sqrt(49)); // a == 7

a = b + x;                  // è equivalente a:
      a = static_cast<int>(static_cast<float>(b)+x); → 7

y = b + x;                  // è equivalente a:
      y = static_cast<float>(b)+x; → 7.8
a = b + static_cast<int>(x+y);
      // a=b+static_cast<int>(9.0); → a = 2 + 9 → 11
a = b + static_cast<int>(x) + static_cast<int>(y);
      // a=b+static_cast<int>(5.8)+static_cast<int>(3.2);
            → a = 2 + 5 + 3 → 10
```

Perdita informazione 1/5

```
int varint = static_cast<int>(3.1415);
```

Perdita di informazione:

3.1415 ≠ `static_cast<double>(varint)`

```
long int varlong = 123456789;
short varshort = static_cast<short>(varlong);
```

Sicuro overflow e quindi valore casuale!

(il tipo short non è in grado di rappresentare un numero così grande)

- **Fondamentale**: in entrambi i casi non viene segnalato alcun errore a tempo di compilazione, né a tempo di esecuzione!

Perdita di informazione 2/5

- Supponiamo di aver memorizzato un numero senza cifre dopo la virgola all'interno di un oggetto di tipo **`double`**
- Supponiamo poi di assegnare il valore di tale oggetto di tipo **`double`** ad un oggetto di tipo **`int`** memorizzato su un numero di bit inferiore al numero di bit della mantissa dell'oggetto di tipo **`double`**
- Si potrebbe avere perdita di informazione?

Perdita di informazione 3/5

- Sì
- L'oggetto di tipo **`int`** potrebbe non essere in grado di rappresentare tutte le cifre
  - Ad esempio, supponiamo di poter rappresentare al più 4 cifre in base 10 con un **`int`** e che invece il valore sia 12543.
  -
- In particolare questo implica che il valore sarebbe numericamente troppo elevato, quindi per l'esattezza si avrebbe un *overflow*
  - Nel precedente esempio numerico, 12543 sarebbe più grande del massimo intero rappresentabile

Perdita di informazione 4/5

- Facciamo invece l'esempio contrario: supponiamo che sia il tipo `int` ad essere memorizzato su un numero di bit **maggiore** del numero di bit utilizzati per rappresentare la mantissa di un oggetto di tipo, per esempio, `float`
- Supponiamo però che, grazie all'uso dell'esponente, il tipo `float` sia in grado di rappresentare numeri più grandi di quelli rappresentabili con il tipo `int`
- In questo caso, si potrebbe avere perdita di informazione se si assegna il valore memorizzato nell'oggetto di tipo `int` all'oggetto di tipo `float`?

Perdita di informazione 5/5

- Sì
- L'oggetto di tipo `float` potrebbe non essere in grado di rappresentare tutte le cifre
- Questo non implica che il valore sarebbe numericamente troppo elevato, quindi non si avrebbe *overflow*
  - Si avrebbe semplicemente un **troncamento delle cifre del numero**
  - Ad esempio, considerando che il tipo `float` può rappresentare al più 6 cifre decimali diverse ed il numero fosse 1412332, sarebbe memorizzato come .141233e7, perdendo l'ultima cifra

Morale

- Le conversioni sono praticamente sempre pericolose
- Quando le si usa bisogna sapere quello che
- si fa
- L'elevata precisione dei moderni tipi numerici fa comunque sì che i fenomeni di perdita di informazione dovuti a cambi di precisione nelle conversioni generino conseguenze serie solo in applicazioni che effettuano elevate quantità di calcoli e/o che necessitano di risultati numerici molto accurati