Corso “Programmazione 1”
Capitolo 02: Variabili, Costanti, Tipi

Docente: Marco Roveri - marco.roveri@unitn.it
Esercitatori: Giovanni De Toni - giovanni.detoni@unitn.it
Stefano Berlato - stefano.berlato-1@unitn.it

C.D.L.: Informatica (INF)
A.A.: 2021-2022
Luogo: DISI, Università di Trento
URL: https://bit.ly/2VgfYwJ

Ultimo aggiornamento: 20 settembre 2021

Terms of Use and Copyright

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Rappresentazione binaria dei numeri

- La rappresentazione binaria dei numeri avviene tramite sequenze di bit (uni e zeri).
- Distinguiamo la rappresentazione per:
  - numeri interi positivi
  - numeri interi con segno
  - numeri reali

Numeri interi positivi

- Una sequenza di bit $b_{n-1}...b_1 b_0$ rappresenta il numero:

$$\sum_{i=0}^{n-1} 2^i \cdot b_i = 2^{n-1} \cdot b_{n-1} + ... + 4 \cdot b_2 + 2 \cdot b_1 + 1 \cdot b_0$$

- Dati n bit è possibile rappresentare numeri nell'intervallo $[0, 2^n - 1]$ (ad esempio, nell'intervallo $[0, 4294967295]$ con 32 bit)
- Esempio (con 8 bit):
  00001001 rappresenta il numero 9 ($2^3 + 2^0 = 8 + 1 = 9$)

Conversione Decimale-Binario

- Serie di divisioni per 2 e analisi del resto
- Convertire 728_{10} in binario

728	0
364	0
182	0
91	1
45	1
22	0
11	1
5	1
2	0
1	1

Risultato: 1011011000_2

Prova: $2^9 + 2^7 + 2^6 + 2^4 + 2^3$
$= 512 + 128 + 64 + 16 + 8 = 728$

Conversione Decimale-Binario

- Convertire 3249_{10} in binario

3249	1
1624	0
812	0
406	0
203	1
101	1
50	0
25	1
12	0
6	0
3	1
1	1

Risultato: 110010110001

Prova: $2^{11} + 2^{10} + 2^7 + 2^5 + 2^4 + 2^0$
$= 2048 + 1024 + 128 + 32 + 16 + 1 = 3249$

Esempi: operazioni tra interi positivi

```
 95 +     01011111       64+16+8+4+2+1          0 + 0 = 0
 54       00110110       32+16+4+2              1 + 0 = 1
-------------------------------------          1 + 1 = 0 (riporto 1)
149       10010101       128+16+4+1             0 + 1 = 1

149 -     10010101       128+16+4+1             0 - 0 = 0
095       01011111       64+16+8+4+2+1          1 - 0 = 1
-------------------------------------          1 - 1 = 0
 54       00110110       32+16+4+2              0 - 1 = 1 (prestato 1
                                                          col. sin.)
```

N.B.: Se eccede il range, il bit di riporto si perde (overflow) $\Longrightarrow$ aritmetica modulo 2^N

```
223 +     11011111       128+64+16+8+4+2+1
 54       00110110       32+16+4+2
-------------------------------------
 21       00010101       16+4+1   (223+54-256=21)
```

Interi con segno

- I numeri interi con segno sono rappresentati tramite diverse codifiche
- Le codifiche più usate sono:
  - codifica segno-valore
  - codifica complemento a 2

Codifica Segno-valore

- Il primo bit rappresenta il segno, gli altri il valore
  - Esempio (8 bit): 10001001 rappresenta -9
- Comporta una doppia rappresentazione dello zero:
  00000000 e 10000000
- I numeri rappresentati appartengono all'intervallo $[-2^{n-1}+1, 2^{n-1}-1]$
  (ad esempio $[-2147483647, 2147483647]$ con 32 bit)
- Poco usato in pratica

Nota:

Occorre usare due diversi algoritmi per la somma a seconda che il segno degli addendi sia concorde o discorde

Esempi: operazioni in segno-valore

Se il segno è diverso, si stabilisce il maggiore dei due in valore assoluto, e si calcolano somme o differenze.

```
         v
149 +    010010101      128+16+4+1          0 - 0 = 0
-95      101011111      64+16+8+4+2+1       1 - 0 = 1
-----------------------------------         1 - 1 = 0
 54      000110110      32+16+4+2           0 - 1 = 1 (prestato 1
         ^                                             col. sin.)
```

Codifica Complemento a 2

- Di gran lunga la più usata
- Un numero negativo è ottenuto calcolando il suo complemento (si invertono zeri e uni) e poi aggiungendo 1
- I numeri rappresentati appartengono all'intervallo $[-2^{n-1}, 2^{n-1} - 1]$ (ad es. $[-2147483648, 2147483647]$ con 32 bit)
- Rappresentazione ciclica: $-X$ in complemento a 2 si scrive come si scriverebbe $2^n - X$ nella rappresentazione senza segno
- Comporta un'unica rappresentazione dello zero: 00000000

Esempio:

- 11110111 in complemento a 2 rappresenta -9:

```
   9       compl.      +1
00001001 11110110 11110111
```

- 11110111 in rappresentazione senza segno rappresenta 247, cioè $2^8 - 9$

Interi senza segno vs. complemento a 2

Codifica	senza segno		complemento a 2	
00000000		0		0
00000001		1		1
⋮		⋮		⋮
00001001	$8+1=$	9	$8+1=$	9
⋮		⋮		⋮
01111111	$64+\ldots+2+1=$	127	$64+\ldots+2+1=$	127
10000000		128	$128-256=$	-128
10000001	$128+1=$	129	$128+1-256=$	-127
⋮		⋮		⋮
10001001	$128+8+1=$	137	$128+8+1-256=$	-119
⋮		⋮		...
11111111	$128+\ldots+1=$	255	$128+\ldots+1-256=$	-1

Esempi: Operazioni in Complemento a 2

```
   -9 +    11110111
    9      00001001
=   0      00000000

   -9 +    11110111
    8      00001000
=  -1      11111111    => 11111110 => 00000001

   -9 +    11110111
   10      00001010
=   1      00000001

   -9 +    11110111
   -5      11111011
=  -14     11110010   => 11110001 => 00001110
```

Conversione da Decimale con Virgola a Binario con Virgola

- Anche i numeri binari si possono esprimere con la virgola!
- Convertire 3.5_{10} in binario con virgola
  - Parte intera: $3_{10} \rightarrow 011_2$
  - Parte frazionaria: $0.5_{10} \rightarrow 2^{-1} \rightarrow 0.1_2$
  - Risultato: 11.1_2
- Prova inversa:
  - $11_2 \rightarrow 2^1 + 2^0 \rightarrow 2 + 1 \rightarrow 3_{10}$
  - $0.1_2 \rightarrow 2^{-1} \rightarrow 0.5_{10}$
  - Risultato: $3 + 0.5 = 3.5_{10}$

Numeri Reali

Rappresentazione in virgola mobile (standard IEEE 754):

- $s \cdot m \cdot 2^{e-o}$, s è il segno ($\{-1, +1\}$), m è la mantissa, e è l'esponente e o è l'offset

Formato	*segno*	*esponente*	*mantissa*	*offset*	*range*	*precisione*
32 *bit*	1 *bit*	8 *bit*	23 *bit*	$2^7 - 1$	$\approx [10^{-38}, 10^{38}]$	$\approx$ 6 *cifre dec.*
64 *bit*	1 *bit*	11 *bit*	52 *bit*	$2^{10} - 1$	$\approx [10^{-308}, 10^{308}]$	$\approx$ 16 *cifre dec.*
128 *bit*	1 *bit*	15 *bit*	112 *bit*	$2^{14} - 1$	$\approx [10^{-4932}, 10^{4932}]$	$\approx$ 34 *cifre dec.*

(nota pratica: $2^{10} = 1024 \approx 1000 \implies 2^{10 \cdot N} \approx 10^{3 \cdot N}$)

- Per la mantissa uso di codifica binaria decimale
  Es: "`.100110011001...`" significa "$1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} + ...$"
  nota: (rappresentazione normale) primo bit assunto essere 1:
  Es: "000100001..." significa "1.000100001..."
- La codifica dell'esponente è dipendente dal numero di bit e dalla particolare rappresentazione binaria (tipicamente codifica binaria senza segno)
- Richiede algoritmi ad-hoc per eseguire le operazioni

Conversione IEEE754

- Convertire il numero -10.75_{10} in floating point a 32 bit (singola precisione)

10	0
5	1
2	0
1	1

0.75	$\downarrow$
0.50	1 (sottraggo 1)
0.	1 (sottraggo 1)
0.	

- Quindi $-1010.11_2 \rightarrow -1.01011_2 * 2^3$ in notazione scientifica
- $(-1)^s * (1 + \textit{Mantissa}) * 2^{\textit{Esponente}-127}$
  - $s = 1$
  - Mantissa $= 1.01011 - 1 = 0.01011$
  - $3 = (\textit{Esponente} - 127)$ quindi Esponente = 130 $\rightarrow$ 10000010

s	Esp. 8bit	Mantissa 23 bit
1	10000010	01011000000000000000000

Conversione IEEE754

- Convertire il numer 0.1875_{10} in floating point a 32 bit (singola precisione)

$0 \mid 0$

0.1875	$\downarrow$
0.3750	0
0.7500	0
0.5000	1 (sottraggo 1)
0.0000	1 (sottraggo 1)

- Quindi $0.0011 \rightarrow 1.1_2 * 2^{-3}$ in notazione scientifica
- $(-1)^s * (1 + \textit{Mantissa}) * 2^{\textit{Esponente}-127}$
  - $s = 0$
  - Mantissa $= 1.1 - 1 = 0.1$
  - $-3 = (\textit{Esponente} - 127)$ quindi Esponente = 124 $\rightarrow$ 01111100

s	Esp. 8bit	Mantissa 23 bit
0	01111100	10000000000000000000000

Conversione IEEE754

- Convertire il numero IEEE754 $0x427d0000_{16}$ in decimale virgola mobile
  - $0x427d0000_{16}$ =0100 0010 0111 1101 0000 0000 0000 0000_2
    $0 \mid 10000100 \mid 11111010000000000000000_2$
  - $N = (-1)^s * (1 + Mantissa) * 2^x$ dove $x = Esponente - 127$
    - $s = 0$
    - Esponente = $10000100 = 2^7 + 2^2 = 132$
      $x = Esponente - 127 = 132 - 127 = 5$
    - Mantissa = $11111010000000000000000 = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-7} = 0.9765625$
    - $N = (-1)^0 * (1 + 0.9765625) * 2^5 = 1 * 1.9765625 * 32 = 63.25$

I Tipi

- Oggetti dello stesso tipo
  - utilizzano lo stesso spazio in memoria e la stessa codifica
  - sono soggetti alle stesse operazioni, con lo stesso significato
- Vantaggi sull'uso dei tipi:
  - correttezza semantica
  - efficiente allocazione della memoria dovuta alla conoscenza dello spazio richiesto in fase di compilazione

Tipi Fondamentali e Derivati in C++

Nel C++ i tipi sono distinti in:

- i tipi fondamentali
  - che servono a rappresentare informazioni semplici
  - Esempio: i numeri interi o i caratteri (int, char, ...)
- i tipi derivati
  - permettono di costruire strutture dati complesse
  - si costruiscono a partire dai tipi fondamentali, mediante opportuni costruttori (array, puntatori, ...)

I Tipi Fondamentali in C++

- i tipi interi: `int`, `short`, `long`, `long long`
- i tipi Booleani: `bool`
- i tipi enumerativi: `enum`
- il tipo carattere: `char`
- i tipi reali: `float`, `double`, `long double`

I primi quattro sono detti tipi discreti (hanno un dominio finito)

I tipi interi (con segno)

- I tipi interi (con segno) sono costituiti da numeri interi compresi tra due valori estremi, a seconda dell'implementazione
  - tipicamente codificati in complemento a 2 con N bit (N = 16, 32, ...)
  - appartengono all'intervallo $[-2^{N-1}, 2^{N-1} - 1]$
- Quattro tipi, in ordine crescente di dimensione
  `short [int]`, `int`, `long [int]`, `long long [int]`
- Dimensioni dipendenti dall'implementazione (macchina, s.o., ...)
  - sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) $\leq$ sizeof(long long)
    (sizeof() restituisce la dimensione del tipo in byte)
  - `short` tipicamente non ha più di 16 bit: $[-32768, 32767]$
  - `long` tipicamente ha almeno 32 bit: $[-2147483648, 2147483647]$

Il valore di un'espressione intera non esce dal range $[-2^{N-1}, 2^{N-1} - 1]$ perché i valori "ciclano": il successore di 2147483647 è -2147483648

Esempio di short, int, long, long long:
{ `../ESEMPI_BASE/shortlong.cc` }

I tipi interi (senza segno)

- Il tipo `unsigned ...` rappresenta numeri interi non negativi di varie dimensioni
  - codifica interi senza segno a N bit (N=16,32, ...), range $[0, 2^N - 1]$
  - tipicamente usati molto poco, come rappresentazioni di sequenze di bit (applicazioni in elettronica)

```
unsigned int x=1232;
unsigned short int x=567;
unsigned long int x=878678687;
```

Il valore di un'espressione unsigned non esce mai dal range $[0, 2^N - 1]$ perché, quando il valore aritmetico X esce da tale range, il valore restituito è X modulo 2^N.

Esempio di unsigned:
{ `../ESEMPI_BASE/unsigned.cc` }

Operatori aritmetici sugli interi

operatore binario (infisso)	significato
`+`	addizione
`-`	sottrazione
`*`	moltiplicazione
`/`	divisione intera
`%`	resto della div. intera

operatore unario (prefisso)	significato
`-`	inversione di segno

Nota:

La divisione è la divisione intera: `5/2` è `2`, non `2.5`!

Esempio sugli operatori aritmetici sugli interi:
{ `../ESEMPI_BASE/operatori_aritmetici.cc` }

Operatori bit-a-bit (su interi senza segno)

operatore	esempio	significato
`>>`	`x>>n`	shift a destra di n posizioni
`<<`	`x<<n`	shift a sinistra di n posizioni
`&`	`x&y`	AND bit a bit tra x e y
`|`	`x|y`	OR bit a bit tra x e y
`^`	`x^y`	XOR bit a bit tra x e y
`~`	`~x`	NOT, complemento bit a bit

Nota:

- `~`: restituisce un intero signed anche se l'input è unsigned

Operatori bit-a-bit : esempio

Siano x e y rappresentati su 16 bit

```
x:     0000000000001100   (12)
y:     0000000000001010   (10)
x|y:   0000000000001110   (14)
x&y:   0000000000001000   ( 8)
x^y:   0000000000000110   ( 6)
~x:    1111111111110011   (65523 oppure -13 )
x>>2:  0000000000000011   ( 3)
x<<2:  0000000000110000   (48)
```

Esempio sugli operatori bit-a-bit su interi senza segno:
{ ../ESEMPI_BASE/operatori_bitwise.cc }