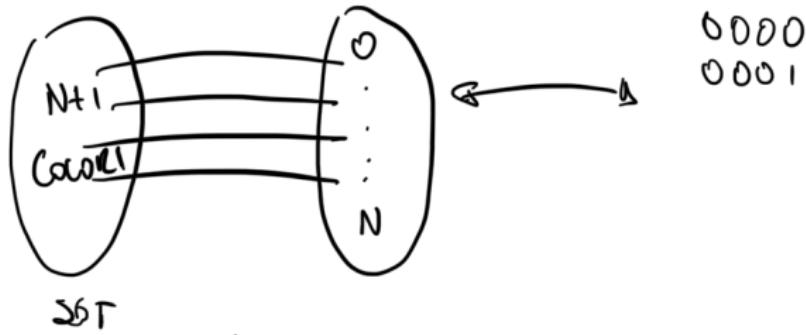


Variabili in C++

Letterale	Tipo	Note
-6	int	int non ha parte frazionaria, può essere negativo
0.5	double	Viene rappresentato in memoria come un double
0.5f	float	Viene rappresentato in memoria come un float
1E6	double	Notazione esponenziale. Eqauivale a 1×10^6 oppure 1000000
10,456	#	Errore in fase di compilazione! Va usato il punto, non la vigola..
<u>3 1/2</u>	#	Errore in fase di compilazione! Va usata una espressione in forma decimale: <u>3.5</u>



Sistema di numerazione in base 2



Sistemi di numerazione posizionale

Nei sistemi di numerazione posizionale, i simboli assumono valori diversi in base alla posizione che occupano nella notazione.

Esempio

$$\begin{aligned} 102_{10} &= 1 \times 10^2 + 0 \times 10^1 + 2 \times 10^0 \\ 10101010_2 &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \\ &\quad 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 170_{10} \\ 8\text{bit} = \text{byte} \end{aligned}$$

Il sistema binario è un sistema di numerazione posizionale in base 2, mentre quello decimale è un sistema di numerazione posizionale in base 10.

Sistema numerico binario

Conversione da base 2 a base 10. Numeri interi

$$\begin{aligned}10101010_2 &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \\&+ 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 170_{10}\end{aligned}$$

Da base 2 a base 10. Numeri con parte frazionaria

Conversione in base 10 del numero 101.0101₂

$$\begin{aligned}\overbrace{\underline{101}}^{2^2} &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = \boxed{5_{10}} \\ \overbrace{\underline{0101}}^{2^{-1} 2^{-2} 2^{-3} 2^{-4}} &= 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = \\ \underline{\underline{1234}} &= \underline{\underline{0.3125_{10}}} \quad \left(\frac{1}{2^4} = \frac{1}{16}\right)\end{aligned}$$

Sistema numerico binario

Conversione da base 10 a base 2. Numeri interi

$$136_{10} = \underline{1} \underline{0} \underline{0} \underline{0} \underline{1} \underline{0} \underline{0} _2$$

$136 : 2 =$	68	$r=0$
$68 : 2 =$	34	$r=0$
$34 : 2 =$	17	$r=0$
$17 : 2 =$	8	$r=1$
$8 : 2 =$	4	$r=0$
$4 : 2 =$	2	$r=0$
$2 : 2 =$	1	$r=0$
$1 : 2 =$	0	$r=1$

Sistema numerico binario

Conversione da base 10 a base 2. Numeri interi

$136_{10} = \mathbf{10001000}_2.$	$136 : 2 = 68 \quad r=0$
	$68 : 2 = 34 \quad r=0$
	$34 : 2 = 17 \quad r=0$
	$17 : 2 = 8 \quad r=1$
	$8 : 2 = 4 \quad r=0$
	$4 : 2 = 2 \quad r=0$
	$2 : 2 = 1 \quad r=0$
	$1 : 2 = 0 \quad r=1$

I resti della divisione **in ordine inverso** costituirano la codifica binaria.

Sistema numerico binario

Conversione da base 10 a base 2. Numeri interi

Sistema x 2⁰

$136_{10} = \underline{10001000}_2$

$\begin{array}{r} 136 \\ - 16 \\ \hline 56 \\ - 48 \\ \hline 8 \\ - 8 \\ \hline 0 \end{array}$

$2^7 = 128$

136 : 2	=	68	r=0
68 : 2	=	34	r=0
34 : 2	=	17	r=0
17 : 2	=	8	r=1
8 : 2	=	4	r=0
4 : 2	=	2	r=0
2 : 2	=	1	r=0
1 : 2	=	0	r=1

Il resto ottenuto dalla **prima divisione** occuperà **l'ultima posizione** nella codifica (bit **meno significativo** o più a destra), e così via . . .

Sistema numerico binario

Conversione da base 10 a base 2. Numeri interi

$$136_{10} = \mathbf{10001000}_2.$$

136 : 2	=	68	r=0
68 : 2	=	34	r=0
34 : 2	=	17	r=0
17 : 2	=	8	r=1
8 : 2	=	4	r=0
4 : 2	=	2	r=0
2 : 2	=	1	r=0
1 : 2	=	0	r=1

... Infine il resto ottenuto **dall'ultima divisione**, che occuperà la **prima posizione** nella codifica (bit **più significativo** o più a sinistra).

Sistema numerico binario

Da base 10 a base 2. Numeri con parte frazionaria

Il numero 28.125₁₀ = 11100.001₂

Parte intera (si converte nel solito modo): 28₁₀ = 11100

Parte frazionaria:

<u>0.125</u> × 2	=	<u>0.250</u>	riporto <u>0</u>
<u>0.250</u> × 2	=	<u>0.500</u>	riporto <u>0</u>
<u>0.500</u> × 2	=	<u>1.000</u>	riporto <u>1</u>
<u>0.000</u> × 2	=	<u>0.000</u>	FINE

0.125₁₀ = **0.001**₂

Sistema numerico binario

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $28_{10} = 11100$

Parte frazionaria:	$0.125 \times 2 = 0.250$	riporto 0
	$0.250 \times 2 = 0.500$	riporto 0
	$0.500 \times 2 = 1.000$	riporto 1
	$0.000 \times 2 = 0.000$	FINE

$0.125_{10} = 0.001_2$

I riporti costituiranno la codifica binaria della parte frazionaria.

Sistema numerico binario

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $28_{10} = 11100$

Parte frazionaria:	$0.125 \times 2 = 0.250$	riporto 0
	$0.250 \times 2 = 0.500$	riporto 0
	$0.500 \times 2 = 1.000$	riporto 1
	$0.000 \times 2 = 0.000$	FINE

$0.125_{10} = 0.001_2$

Il riporto ottenuto dalla prima moltiplicazione occuperà la **prima posizione** nella codifica (**bit più significativo**), e così via ...

Sistema numerico binario

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $28_{10} = 11100$

Parte frazionaria:	$0.125 \times 2 = 0.250$	riporto 0
	$0.250 \times 2 = 0.500$	riporto 0
	$0.500 \times 2 = 1.000$	riporto 1
	$0.000 \times 2 = 0.000$	FINE

$0.125_{10} = 0.001_2$

... infine, il riporto ottenuto dall'ultima moltiplicazione occuperà
l'ultima posizione nella codifica (bit **meno significativo**)

Sistema numerico binario

Da base 10 a base 2. Arrotondamento per troncamento

Il numero $\underline{17.55}_{10} = \underline{10001.10001100}_2$ (*)

00000000

Parte intera: $\underline{17}_{10} = \underline{10001}$

Parte frazionaria:

$\underline{0.55} \times 2$	=	<u>1.10</u>	riporto <u>1</u>
$\underline{0.10} \times 2$	=	0.20	riporto <u>0</u>
$\underline{0.20} \times 2$	=	0.40	riporto <u>0</u>
$\underline{0.40} \times 2$	=	0.80	riporto <u>0</u>
$\underline{0.80} \times 2$	=	1.60	riporto <u>1</u>
$\underline{0.60} \times 2$	=	1.20	riporto <u>1</u>
$\underline{0.20} \times 2$	=	0.40	riporto <u>0</u>
$\underline{0.40} \times 2$	=	0.80	riporto <u>0</u>

Sistema numerico binario

Parte frazionaria:

0.55×2	=	1.10	riporto 1
0.10×2	=	0.20	riporto 0
0.20×2	=	0.40	riporto 0
0.40×2	=	0.80	riporto 0
0.80×2	=	1.60	riporto 1
0.60×2	=	1.20	riporto 1
0.20×2	=	0.40	riporto 0
0.40×2	=	0.80	riporto 0
...	=

(*) Arrotondamento per **troncamento** alla ottava cifra:

$$0.55_{10} = 0.10001100_2, \Rightarrow 17.55_{10} = 10001.10001100$$

Sistema numerico binario

Nel precedente esempio si è ottenuta una codifica binaria *parziale* del numero 0.55_{10} .

Bisognerebbe avere a disposizione più bit! Quanti? Infiniti!

Infatti alcuni numeri (con parte frazionaria) **non si possono rappresentare con un numero finito di cifre!**

Ciò accade sia nel sistema decimale, che nel sistema binario:

Ci vorrebbero infiniti bit!

$$\text{Base } 10: \frac{1}{3} = 0.3333\dots 3\dots = 0.\overline{3}.$$

$$\text{Base } 2: 4.35_{10} = 100.0101100\dots 1100\dots = 100.010\overline{1100}$$

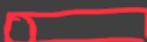
Rappresentazione dei numeri al calcolatore: standard IEEE 754

Rappresentazione dei numeri nei calcolatori

Tipicamente, per la rappresentazione dei numeri nei calcolatori si impiegano **sequenze di bit di lunghezza variabile** (8, 16, 32, 64, ...).

$$2^8 = \underline{256} \text{ NUMERI INTERI SENZA SEGNO } 0 - 255$$

Numeri interi



0 - 255

Per rappresentare gli **interi (con o senza segno)**, i bit si impiegano per rappresentare:

- il **valore assoluto** (o modulo) del numero stesso.
- Eventuale **segno**. In questo caso si “perde” un bit: il range di valori rappresentabili, in modulo, è dimezzato rispetto alla rappresentazione senza segno.

Rappresentazione dei numeri nei calcolatori

Numeri interi senza segno

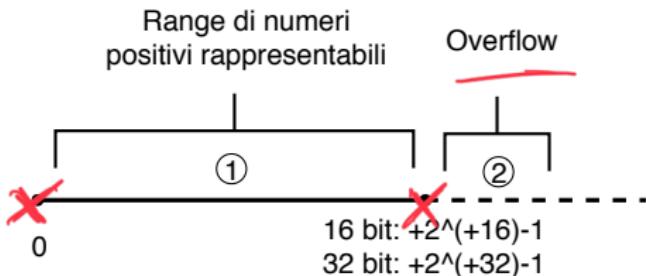
Codifica di Numeri interi **senza segno** con 16 bit:

Intervallo numerico: $[0, 2^{16} - 1] = [0, \underline{65.535}]$ 65536

Codifica di numeri interi **senza segno** con 32 bit:

Intervallo numerico: $[0, 2^{32} - 1] = [0, \underline{4294967295}]$

Interi senza segno (16/32 bit)



Rappresentazione dei numeri nei calcolatori

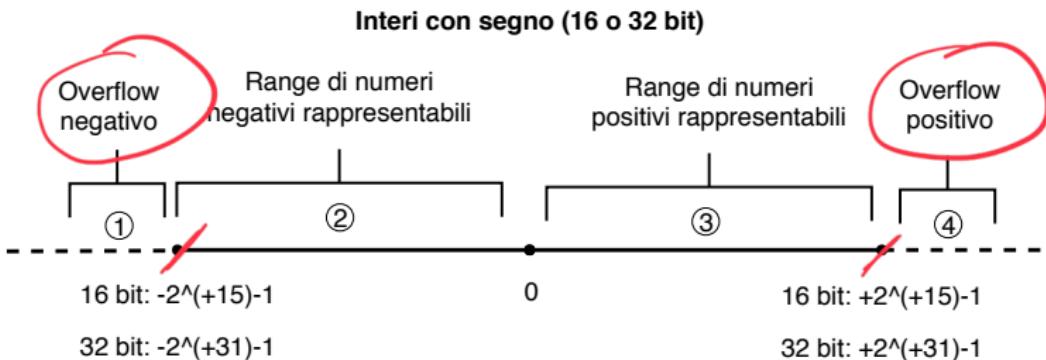
Numeri interi con segno

Codifica di numeri interi **con segno** a 16 bit:

Intervallo numerico $\pm(2^{15} - 1) = \underline{32767}$ (1 bit per il segno)

Codifica di numeri interi **con segno** a 32 bit:

Intervallo numerico $\pm(2^{31} - 1) = 2.147.483.647$ (1 bit per il segno)



Rappresentazione dei numeri nei calcolatori

Numeri reali

Nel calcolatore, i **numeri reali** sono rappresentati mediante un formato detto in **virgola mobile** (floating point).

Si tratta di una rappresentazione in forma compatta che deriva dalla rappresentazione scientifica.

Esempio in base 10

a) 96.103 = 0.96103 $\times 10^{+2}$

b) 2.96 = 0.296 $\times 10^{+1}$

c) 2.96 = 29.6 $\times 10^{-1}$

Rappresentazione dei numeri nei calcolatori

Numeri reali

Esempio in base 10

- a) $96.103 = 0.96103 \times 10^{+2}$
- b) $2.96 = 0.296 \times 10^{+1}$
- c) $2.96 = 29.6 \times 10^{-1}$

0.96103, 0.296 e 29.6 sono denominati **Mantissa o significando**.

10 è la **base**.

+2 e +1 e -1 sono denominati **esponente**.

Esempio in base 10

a) $96.103 = 0.96103 \times 10^{+2}$

b) $2.96 = 0.296 \times 10^{+1}$

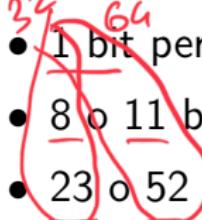
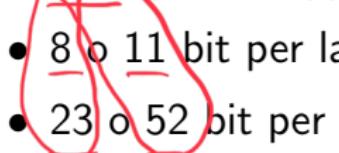
c) $2.96 = 29.6 \times 10^{-1}$

Osservazione: i numeri b) e c) sono uguali, cambia solo la posizione della virgola, che si dice “flottante”, ecco perchè il termine floating point.

Rappresentazione dei numeri nei calcolatori

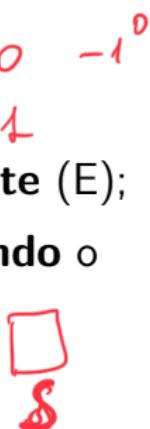
Standard IEEE 754 per i calcoli in virgola mobile

Lo standard IEEE 754 definisce il formato per la rappresentazione dei numeri in virgola mobile:

- 1 bit per la rappresentazione del segno (s); 
- 8 o 11 bit per la rappresentazione dello esponente (E); 
- 23 o 52 bit per la rappresentazione del significando o mantissa (M); 

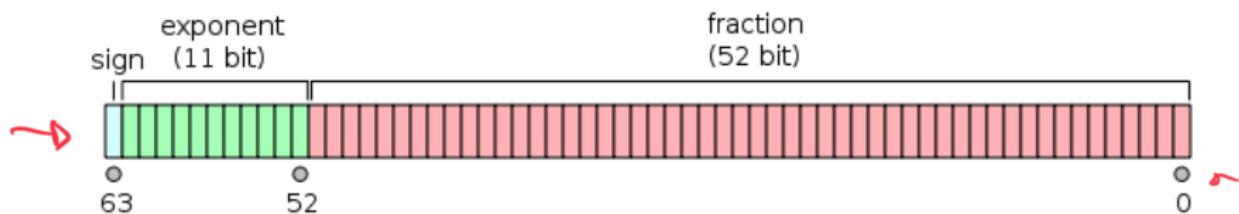
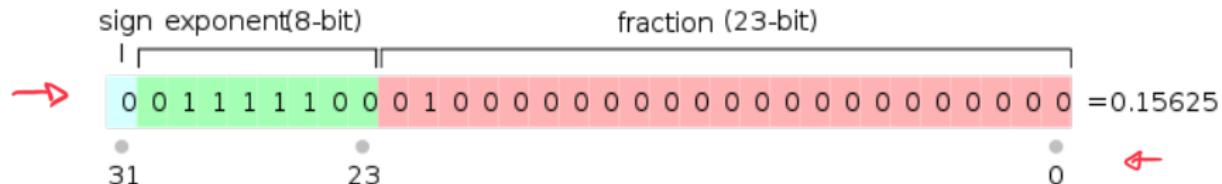
Tali che:

$$N = \underline{(-1)^s} \times \underline{2^E} \times \underline{M}$$



Rappresentazione dei numeri nei calcolatori

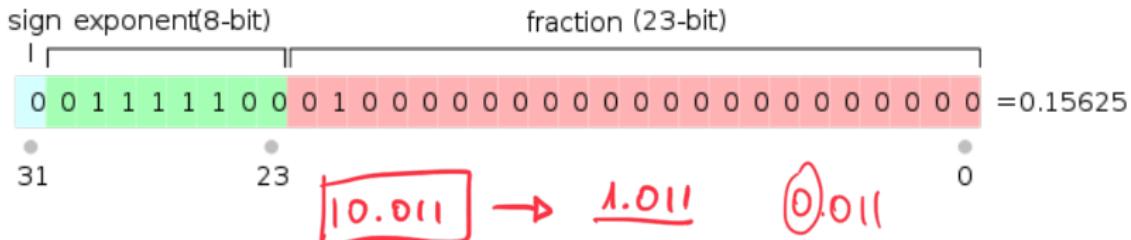
IEEE 754



NB: "fraction" è la mantissa o significando.

Rappresentazione dei numeri nei calcolatori

Standard IEEE 754: Calcolo e rappresentazione della mantissa.



Mantissa o significando rappresentati in forma **normalizzata**:

- si moltiplica o si divide la codifica binaria della mantissa per una certa potenza di 2.
 - In tal modo rappresentazione della mantissa rimarrà solo una cifra prima della virgola, cioè 1.
 - Inoltre, dato che la cifra prima della virgola è sempre 1, questa non viene rappresentata, risparmiando così 1 bit.

Rappresentazione dei numeri nei calcolatori

Standard IEEE 754: Calcolo e rappresentazione della mantissa.

Esempio

Calcolo della mantissa o significando in formato **IEEE 754** a singola precisione (23 bit) per il numero -113.25_{10}

1. Si calcola la **codifica** in base 2 del **valore assoluto** del numero: $\underline{113.25}_{10} = 1110001.01_2$.
2. Per **normalizzare**, spostiamo la virgola di 6 posti :

$$\underline{\underline{1110001.01}} = \underline{1.11000101} \times 2^{\underline{-6}}$$

Standard IEEE 754: Calcolo e rappresentazione della mantissa.

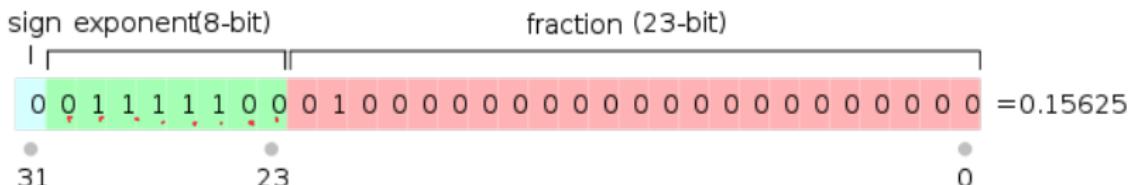
Esempio

Calcolo della mantissa o significando in formato IEEE 754 a singola precisione (23 bit) per il numero -113.25_{10}

3. Ricordando che **la cifra alla sinistra delle virgola non si rappresenta**, la mantissa sarà costituita dai segg. 23 bit:
 - Bit **1-8** (dal più significativo): 11000101
 - Bit **9-23**: (fino al bit meno significativo): 00000000000000000000
4. Quindi sarà $M = 1100010100000000000000000$

Rappresentazione dei numeri nei calcolatori

Standard IEEE 754: Calcolo e rappresentazione dello esponente



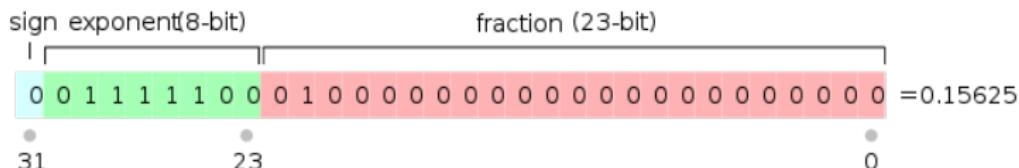
Con 8 bit, in teoria si avrebbero 256 combinazioni.

In pratica:

- I valori 0 e 255 sono **riservati per usi speciali** (se ne parlerà dopo).
- In pratica **si rappresentano 254 valori**, da -126 a +127.

Rappresentazione dei numeri nei calcolatori

Standard IEEE 754: Calcolo e rappresentazione dello esponente

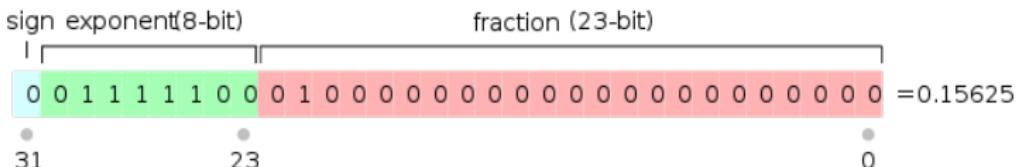


Infine, il valore dello esponente si rappresenta a meno di un valore k detto **bias**:

- $E = \underline{e} + \underline{k}$.
- E è il valore **effettivamente rappresentato** nel campo esponente.
- k è il **bias**.
- e è il valore **esponente** ottenuto durante il **calcolo della mantissa**.

Rappresentazione dei numeri nei calcolatori

Standard IEEE 754: Calcolo e rappresentazione dello esponente



Nel caso dei floating point a 32 bit si ha $k = +127$,

Dunque se $-126 \leq e \leq +127$ ed $E = e + k$ allora $1 \leq E \leq 254$

In tal modo:

- (+) Le codifiche 0 e 255 si possono riservare per usi speciali.
- (+) Non si è costretti a rappresentare il segno per il campo esponente (in quanto E non ha segno). Rappresentare il segno darebbe problemi nel confronto tra numeri.

Rappresentazione dei numeri nei calcolatori

Standard IEEE 754: Calcolo e rappresentazione dello esponente

Nello ESEMPIO precedente era

$$113.25_{10} = 1110001.01_2 = 1.11000101 \times 2^6$$

Quindi $e = 6$.

$$\text{Allora } E = 6 + k = 6 + 127 = 133 = 10100001.$$

Infine, il **segno**: dato che il numero -113.25 è negativo, $s=1$.

Quindi la codifica a 32 bit floating point IEEE 754 del numero -113.25 è la seguente:

$s \quad E \quad M$

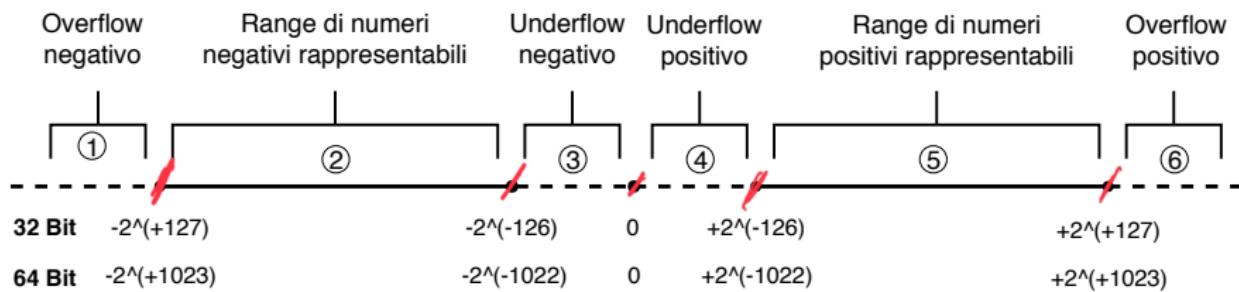
| 1 | 10100001 | 110001010000000000000000 | 

Rappresentazione dei numeri nei calcolatori

Floating point IEEE 754: intervalli numerici

NB: Gli intervalli 2 e 5 sono costituiti da **un numero finito di elementi** che costituiscono un sottoinsieme di \mathbb{R} .

IEEE 754 singola precisione (32 e 64bit)



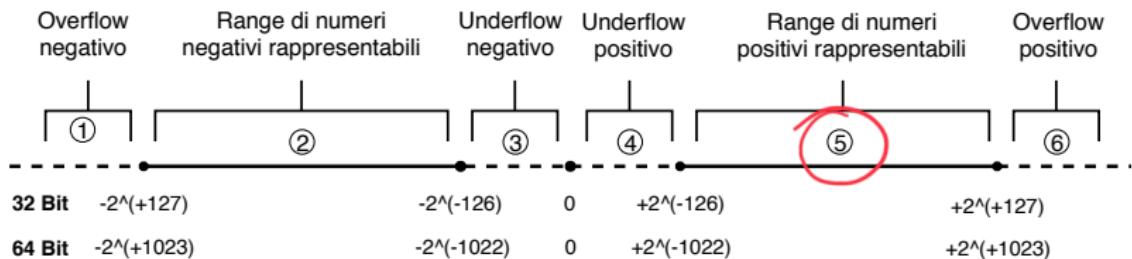
Approssimazioni

1) **Valore non rappresentabile con un numero finito di cifre**, la sua rappresentazione sarà **troncata**.

ESEMPIO: $4.35_{10} = 100.010\overline{1100}$.

Il numero 4.35 ricade all'interno del range dell'intervallo 5, ma non ne fa parte (non è rappresentabile senza approssimazione).

IEEE 754 singola precisione (32 e 64bit)



Approssimazioni

- 2) Valore con un **numero di cifre significative** maggiore del numero massimo rappresentabile nel campo mantissa (o significando).

Esempio

Si consideri il numero **9876543.25**.

Codifica floating point IEEE 754 a singola precisione (32 bit):

$$\begin{aligned} \underline{\underline{9876543.25}}_{10} &= 100101101011010000111111.01 = \\ &= \underline{\underline{1.0010110101101000011111101}} \times 2^{23}. \end{aligned}$$

NB: Mantissa (o significando) di lunghezza $\underline{\underline{25}} > 23$

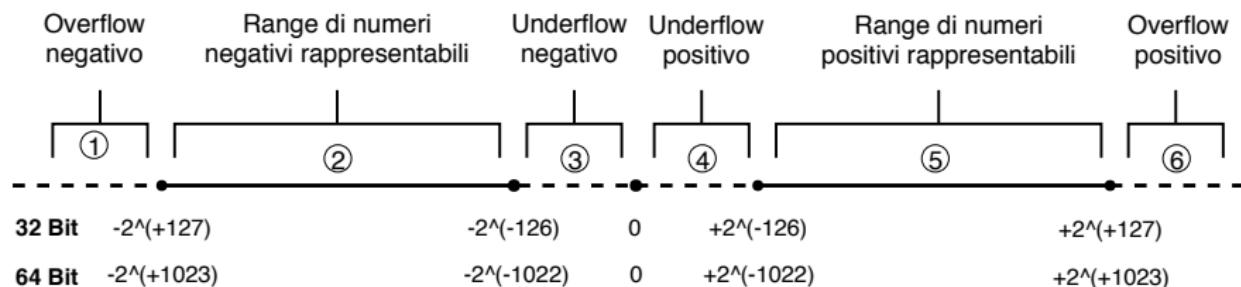
Ma per **floating point 32 bit** lunghezza massima mantissa **23 bit!**.

⇒ il valore sarà memorizzato in modo approssimato!

Approssimazioni

Osservazione: Il numero **9876543.25** rientra, in valore assoluto, nel range dell'intervallo 5 di IEEE 754 a singola precisione.

IEEE 754 singola precisione (32 e 64bit)



Approssimazioni

Precisione di una rappresentazione in virgola mobile

In generale una certa rappresentazione floating point è caratterizzata da **una precisione p**.

La precisione p è costituita dal numero di **cifre significative** che è possibile rappresentare in quel determinato formato.

Esempio

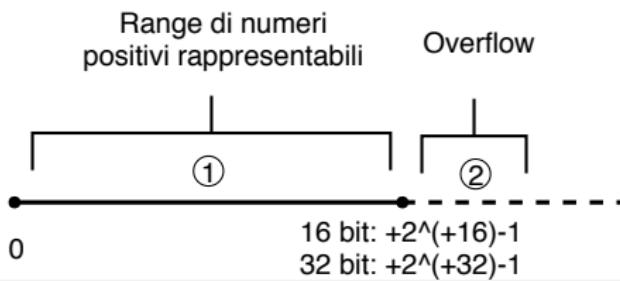
Si consideri il numero **1234.030405887000**₁₀.

Le cifre significative sono costituite dalla sequenza
1234030405887

Overflow: Il risultato numerico di una certa espressione aritmetica è **maggiore**, in valore assoluto, del valore massimo rappresentabile.

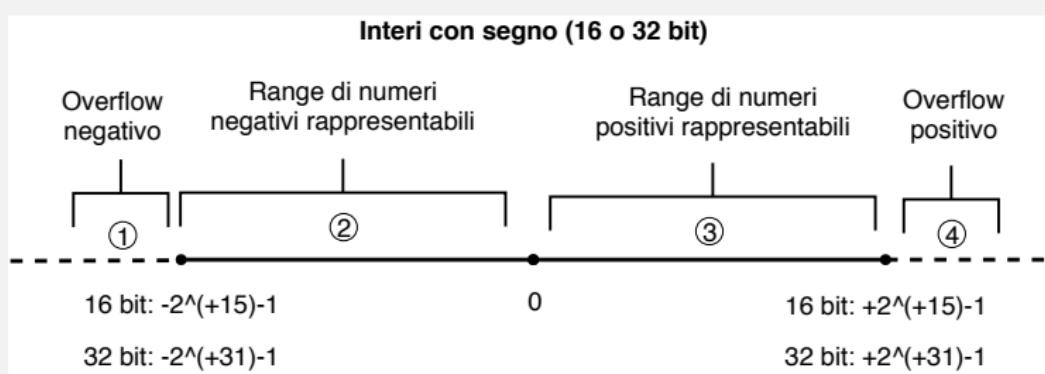
Overflow per interi senza segno

Interi senza segno (16/32 bit)



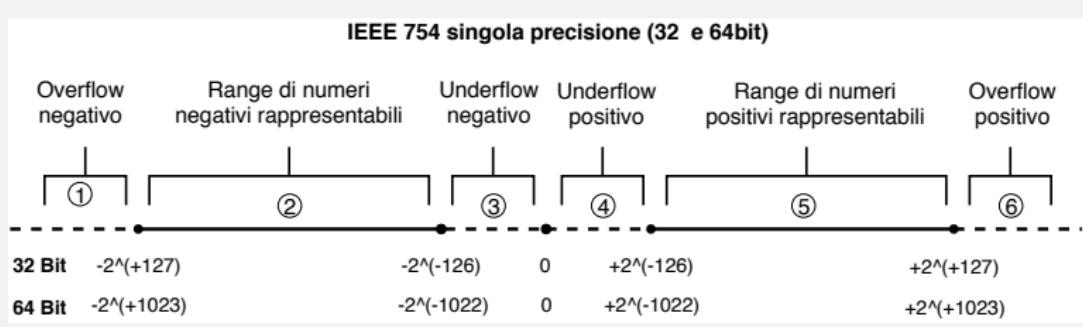
Overflow: Il risultato numerico di una certa espressione aritmetica è **maggiore**, in valore assoluto, del valore massimo rappresentabile.

Overflow per interi con segno



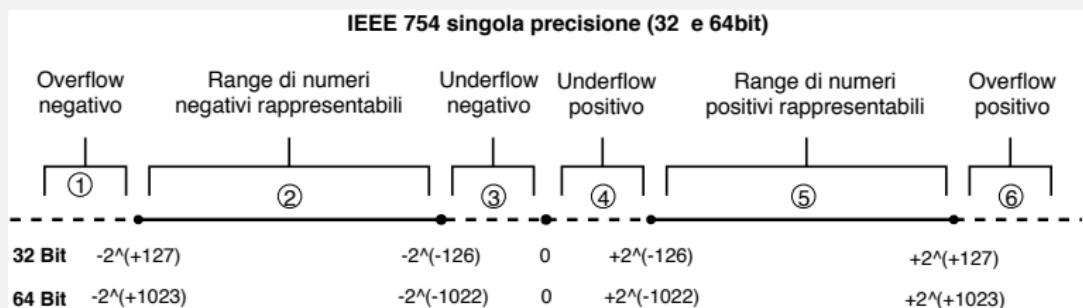
Overflow: Il risultato numerico di una certa espressione aritmetica è **maggiore**, in valore assoluto, del valore massimo rappresentabile.

Overflow per numeri floating point



Underflow

Underflow (Floating point): Il risultato di una operazione tra valori di un certo tipo è **minore**, in valore assoluto, del più piccolo valore rappresentabile.



IEEE 754: valori speciali

Lo standard IEEE 754 riserva alcune combinazioni di bit per la rappresentazione di alcuni **valori speciali**.

Valori Speciali IEEE 754

$\pm\text{Inf}$: Divisione di numeri in virgola mobile per zero oppure casi di **overflow (positivo o negativo)**: $\pm\frac{x}{0}$.

NaN Forme indeterminate (risultato indefinito) : $\frac{0}{0}, \frac{\pm\text{Inf}}{\pm\text{Inf}}$, oppure ancora $+\text{Inf} - \text{Inf}$, e così via.

± 0 I casi di underflow daranno come risultato **zero con segno** a seconda che il segno della espressione sia positivo o negativo. Una certa combinazione di bit permette di rappresentare entrambi gli zeri.

Rappresentazione dei numeri nei calcolatori

Per rappresentare i numeri nei **calcolatori** si usa la **rappresentazione base 2**.

Questa differisce a seconda che si debbano rappresentare **numeri interi** ($n \in \mathbb{Z}$) o numeri **decimali (reali)** ($n \in \mathbb{R}$).

Riassumendo..

Rappresentazione binaria di numeri interi

- Un bit (opzionale) per il **segno**
- **Tutti i numeri all'interno del range specificato** dalla rappresentazione **sono rappresentabili** (NO approssimazione, NO underflow).
- Possibile **OVERFLOW** (negativo o positivo)
- Al fine di **non sprecare spazio in memoria e non incorrere in overflow**, è importante scegliere una opportuna rappresentazione
 1. numero di bit (ES: 16 o 32);
 2. eventuale presenza del bit per il segno.

Numeri floating point (virgola mobile)

- Standard IEEE 754 per singola precisione (32 bit) o doppia precisione (64 bit).
- Codifica/rappresentazione *compatta* mediante **significando** (o mantissa), **segno** ed **esponente**.
- La **precisione** p di una codifica floating point è rappresentata dal **numero di cifre significative** che è possibile rappresentare nella mantissa o significando.
 - 6 cifre per la precisione singola
 - 15 cifre per la precisione doppia

Numeri floating point (virgola mobile)

- **Errori di approssimazione** quando
 - il numero non è rappresentabile con numero finito di cifre.
 - il numero di cifre significative del numero da rappresentare è maggiore della precisione p
- **Underflow** quando il numero da rappresentare, in valore assoluto, è troppo piccolo per essere rappresentato.
- **Overflow**: il valor assoluto del numero è maggiore, del numero più grande (in valore assoluto) rappresentabile con quella codifica.

Tipi in C++

Tipi numerici nel C++

Dimensione e range sono valori tipici!

Tipo	Range (tipico)	Precisione	Dimensione
<u>int</u>	$\pm 2.147.483.647$	–	4 bytes
<u>unsigned</u>	$[0, 4.294.967.295]$	–	4 bytes
<u>long</u>	$\pm (2^{63} - 1)$	–	8 bytes
<u>unsigned long</u>	$[0, 2^{64} - 1]$	–	8 bytes
<u>short</u>	± 32768	–	2 bytes
<u>unsigned short</u>	$[0, 65535]$	–	2 bytes
<u>double</u>	$\pm 10^{308}$	15 cifre	8 bytes
<u>float</u>	$\pm 10^{38}$	6 cifre	4 bytes



La specifica dello standard per il C++ (ISO/IEC 14882:2017) **non definisce in modo completo** il numero di bytes o il range di valori per i tipi numerici.

Tali valori generalmente variano con l'architettura e l'implementazione (compilatore + librerie standard).

1. Sarà importante **conoscere la dimensione** quando si debbono allocare **grandi porzioni di memoria**.

ESEMPIO:

- allocare un array di miliardi di elementi di tipo int
- il fatto che un tipo int occupi 8 bytes piuttosto che 4 bytes sarà determinante. Nel primo caso la memoria potrebbe non bastare.

2. **Conoscere intervallo di valori rappresentabili** dei tipi in virgola mobile per non incorrere in overflow. ESEMPIO: per gli interi con ordine di grandezza 10^{10} useremo i long.

3. **Conoscere la precisione p** dei tipi in virgola mobile per non incorrere in underflow o approssimazioni non previsti.

Tipi numerici nel C++

Numero di byte corrispondenti ad un determinato tipo

In C/C++ esiste l'operatore sizeof(), che si usa come una funzione con argomento il particolare tipo o anche una variabile di un certo tipo.

```
//Stampa la dimensione in byte del tipo int  
cout << sizeof(int) << '\n';  
//Stampa la dimensione in byte del tipo double  
cout << sizeof(double) << '\n';
```

Tipi numerici nel C++

Intervalli numerici e precisione

C++ eredita dal C gli header float.h e limits.h

```
#include <climits> // header da includere  
INT_MIN, INT_MAX // più piccolo/grande valore intero  
LONG_MIN, LONG_MAX // più piccolo/grande valore long
```

```
#include <cfloats> //header da includere  
// più piccolo/grande valore float positivo  
FLT_MIN, FLOAT_MAX  
// no. di cifre significative rappresentabili  
// (precisione!)  
FLT_DIG, DBL_DIG
```

Tipi numerici nel C++

Intervalli numerici e precisione

La libreria standard del C++ include la **classe template numeric_limits**.

```
#include <limits>
cout << std::numeric_limits<int>::lowest();
cout << std::numeric_limits<double>::lowest();
```

Si tratta di un tentativo di uniformare e standardizzare l'interfaccia per la lettura dei range numerici in un sistema.

Documentazione:

https://en.cppreference.com/w/cpp/types/numeric_limits

Esempi svolti

A1_01_types_limits.cpp

Altri tipi in C++

<https://en.cppreference.com/w/cpp/language/types>

Tipo	Range (tipico)	Dimensione
<u>char</u>	[-128,+127]	<u>1 byte</u>
<u>unsigned char</u>	[0,255]	<u>1 byte</u>
<u>bool</u>	{true,false}	<u>1 byte</u>
<u>void</u>	#	<u>1 byte</u>

Il tipo void sta ad indicare **nessun valore di ritorno**.

In C si usano spesso puntatori void (void *) per restituire indirizzi di memoria per i quali il “chiamante” farà un **casting** ad un tipo specifico di puntatore.

Esempi svolti

A1_00_var.cpp

A1_01_types.cpp

A1_02_precision.cpp

A1_03_overflow.cpp

A1_04_inf.cpp

Operatori aritmetici, funzioni matematiche di base, conversioni

Operatori aritmetici

<https://en.cppreference.com/w/cpp/language/expressions>

Operatore	Num. argomenti	Significato
*	2	Moltiplicazione
/	2	Divisione
+	2	Somma
-	2	Sottrazione
++	1	Incremento
--	1	Decremento
%	2	Modulo

Operatore modulo restituisce il **resto della divisione** tra due numeri **interi**.

Esempi svolti

A1_01_unary_op.cpp

Funzioni matematiche (libreria math.h)

<https://en.cppreference.com/w/cpp/header/cmath>

```
#include <cmath> //The C++ way, oppure ...
#include <math.h> //The C way, inoltre ...
using namespace std;//per comodità
```

Funzione	Argomenti	Significato
pow()	(x,y) in virgola mobile	x^y
sqrt()	(x,y) in virgola mobile	\sqrt{x}
sin()	(x) in virgola mobile	$\sin(x)$
abs()	(x) in virgola mobile	$ x $
log()	(x) in virgola mobile	$\ln x$
log10(), log2()	(x) in virgola mobile	$\log_{10}(x)$, $\log_2(X)$

Uso delle funzioni matematiche nei seguenti file.

Esempi svolti

A1_02_precision.cpp

A1_04_inf.cpp

A1_04_constMath.cpp

Tipi numerici: conversioni e promozioni

⚠ Attenzione ai passaggi da un tipo ad un altro che **contiene meno informazioni** (e.g. long \leftarrow int, double \leftarrow int, etc.)

Da floating point a intero

```
double d = 2.8965;  
int a = d; // Perdita della frazione!
```

Arrotondare allo intero più vicino?

```
double d = 2.8965;  
int a = d+0.5; // Arrotondamento OK
```

Tipi numerici: conversioni e promozioni

Divisione / moltiplicazione tra interi

```
float result = 5 / 2; // =2 perdita della frazione!
```

```
float result = 5.0 / 2; // =2.5 OK
```

Quale è la differenza?

Conversioni implicite

Per una espressione che contiene interi e numeri in virgola mobile, basta che uno dei membri della espressione sia **esplícitamente floating point**.

Il compilatore opererà una **conversione implicita** in virgola mobile degli altri membri per eseguire moltiplicazioni e divisioni in floating point, **senza alcuna perdita di informazione**.

Casting statico

Casting o conversione esplicita

Il **casting** è un'operazione con cui si indica al compilatore che **deve convertire** una variabile o il risultato di una espressione ad tipo ben definito.

Si rende necessario quando si vuole una **conversione differente** dalla conversione implicita.

Il C++ prevede uno apposito operatore denominato *static_cast*:

static_cast<type>(value)

Casting statico

Esempio di casting

```
double d = 10.73;  
//arrotonda all'intero più vicino  
double result = static_cast<int> (10 * d + 0.5);  
//ma anche C-style  
double result = (int) (10 * d + 0.5);
```

Esempi svolti

A1_04_casting.cpp

FINE