12/9/25

FLOATING POINT NUMBERS

Quando parliamo dei floating point numbers, non ci riferiamo a semplici numeri “con la virgola”, bensì una rappresentazione binaria molto specifica che consente di descrivere valori reali, anche estremamente grandi e piccoli, all’interno di un numero limitato di bit. Il punto chiave è che questa rappresentazione non è diretta come quella degli interi, e di conseguenza le operazioni su numeri in virgola mobile non garantiscono sempre il risultato esatto che ci aspetteremmo in matematica.
In linguaggi come il C, due numeri floating point che sembrano identici possono non risultare uguali usando l’operatore ==. Questo accade perché il numero memorizzato è spesso solo un’approssimazione del valore reale. Se pensiamo a un intero a 32 bit, i suoi bit rappresentano letteralmente un numero in base due: ogni configurazione corrisponde a un unico intero, e l’aritmetica (a meno di overflow) è esatta.
Un “float” a 32 bit, invece, utilizza i bit in modo più complesso: alcuni sono riservati al segno, altri all’esponente e altri ancora alla mantissa, cioè alla parte frazionaria che definisce le cifre significative. Questo schema, definito dallo standard IEEE 754, permette di coprire intervalli molto ampi, ma riduce il numero di valori rappresentabili in modo preciso.
Per chiarire la differenza fra interi e floating point, introduciamo due concetti di metrologia: **accuratezza** e **precisione**. *L’accuratezza* indica quanto una misura si avvicina al valore vero; *la precisione* misura invece quanto una quantità è “definita” o “dettagliata”. Gli interi sono perfettamente accurati: il numero 2 rappresenta esattamente 2, e 2 + 1 dà esattamente 3. Tuttavia, la loro precisione è limitata: se divido 5 e 4 per 2 ottengo sempre 2, perdendo informazione sulla parte frazionaria. Sono come mattoni: solidi e fedeli, ma incapaci di adattarsi a sfumature sottili.
I floating point sono l’opposto: molto precisi, perché riescono a descrivere gradazioni fini, ma poco accurati. Possiamo pensare a una sostanza modellabile, come il pongo o il “silly putty”: possiamo sagomarla in forme complesse, ma non otterremo mai spigoli perfetti. Questo perché molti numeri razionali non hanno una rappresentazione binaria finita. Un esempio classico è 1/3: in base dieci è 0,3333… all’infinito, e in base due accade lo stesso per frazioni come 1/10. Un numero floating point può solo avvicinarsi, introducendo un inevitabile **errore di arrotondamento**.
Dal punto di vista matematico, possiamo pensare a un float come a un numero del tipo

$$x = (-1)^s \times m \times 2^e$$

dove s è il bit di segno, m la mantissa normalizzata (che porta l’informazione sulle cifre significative) ed e l’esponente che sposta la “virgola” binaria. *Con 32 bit* (singola precisione), ad esempio, *abbiamo 1 bit per il segno, 8 per l’esponente e 23 per la mantissa.* Questa struttura permette di rappresentare numeri molto grandi o molto piccoli, ma sempre con un numero finito di combinazioni, quindi con un insieme discreto di valori possibili.

STANDARD IEEE754

Quando parliamo di numeri in virgola mobile in un calcolatore, ci riferiamo allo standard IEEE 754, il sistema di rappresentazione universalmente adottato per garantire portabilità e coerenza tra hardware e software. L'idea di base è esprimere un numero reale nella forma scientifica in base due, cioè come prodotto di un segno, una mantissa normalizzata e una potenza di due. La formula generale per un numero **normalizzato** è

$$x = (-1)^s \times (1.f)_2 \times 2^{e - bias}$$

dove s è il bit di segno (0 per positivo, 1 per negativo), f rappresenta la frazione binaria ed e l'esponente memorizzato. Il termine *bias* è un valore fisso che permette di codificare esponenti negativi usando interi non negativi: nel formato **binary32** (detto anche *single precision*) l'esponente occupa 8 bit e il bias è 127; nel formato **binary64** (*double precision*) i bit di esponente sono 11 e il bias è 1023.

La mantissa include un "1" implicito, da cui il termine "1.f": questo consente di guadagnare un bit di precisione perché il primo uno è sempre presente nei numeri normalizzati. Per i numeri **subnormali**, che servono a rappresentare valori vicini allo zero, la formula diventa

$$x = (-1)^s \times (0.f)_2 \times 2^{e - bias}$$

senza l'uno implicito. Questa zona subnormale garantisce la cosiddetta gradual underflow, cioè un passaggio regolare verso lo zero. Alcune combinazioni speciali di bit rappresentano valori non numerici. Se tutti i bit di esponente sono a 1 e la frazione è a zero, si ottengono $+\infty$ e $-\infty$ a seconda del segno. Se l'esponente è tutto a 1 e la frazione contiene almeno un bit a 1, si parla di NaN (*Not a Number*), usato ad esempio per risultati di operazioni indeterminate.Il processo di calcolo introduce inevitabilmente errori di arrotondamento. IEEE 754 definisce diverse modalità, la più comune è "round to nearest, ties to even", che sceglie il numero più vicino e, in caso di equidistanza, quello con bit meno significativo pari. Concetti come **ULP** (*Unit in the Last Place*) e **epsilon macchina** descrivono rispettivamente la distanza tra due numeri consecutivi rappresentabili e il più piccolo incremento che, sommato a 1.0, produce un risultato distinguibile da 1.0 stesso.

La dimensione: è l'*esponente*. Dice quante volte dobbiamo moltiplicare o dividere per 2 per rendere il numero grande o piccolo, un po' come spostare la virgola a destra o a sinistra. **La forma precisa**: è la *mantissa*, che contiene le cifre del numero in base 2, cioè scritte con soli 0 e 1.

Un numero IEEE 754 è composto da tre campi: **bit di segno**, **esponente** e **mantissa** (o frazione). Nel formato single precision abbiamo 1 bit di segno, 8 bit di esponente e 23 bit di mantissa. La disposizione è quindi:

[segno | esponente (8 bit) | mantissa (23 bit)]

Alcune combinazioni sono riservate a casi speciali. Se l'esponente è tutto a 1 (255 in single) e la mantissa è zero, il numero rappresenta $+\infty$ o $-\infty$ a seconda del segno. Se l'esponente è tutto a 1 ma la mantissa contiene almeno un bit a 1, si ottiene un **NaN** (*Not a Number*), usato per risultati indefiniti come $0/0$ o $\infty - \infty$. Lo zero è un'eccezione opposta: esponente e mantissa tutti a zero, con segno positivo o negativo.

I limiti numerici derivano direttamente da questa codifica. Per il formato single precision il massimo valore finito è circa 3.402823×1038, il più piccolo numero normalizzato è circa 1.175494×10−38, e il più piccolo subnormale scende fino a 1.401298×10−45. Nel double precision gli intervalli sono molto più ampi: fino a 1.79×10308 e minimi nell'ordine di 10−308.

Un parametro importante è l'**epsilon macchina**: è il più piccolo incremento ε tale che $1+\varepsilon>1$. Vale circa 1.19×10−7 per i float a 32 bit e 2.22×10−16 per i double. Questo numero misura la distanza tra 1 e il successivo numero rappresentabile, cioè l'unità nell'ultima posizione (*Unit in the Last Place*, ULP).

Dal punto di vista della programmazione, queste caratteristiche portano a conseguenze pratiche. Confrontare due float con l'operatore == verifica solo l'uguaglianza bit-per-bit, ma spesso ciò non ha senso perché le operazioni di calcolo introducono inevitabili arrotondamenti. È più corretto confrontare la differenza relativa o in termini di ULP, scegliendo una tolleranza che tenga conto della scala dei numeri.

LA RAPPRESENTAZIONE CON IL PUNTO FISSO – FIXED POINT

Quando si parla di numeri reali in informatica si pensa subito alla virgola mobile e allo standard IEEE 754. Ma la rappresentazione in **punto fisso** (fixed point) resta una soluzione importante, soprattutto in contesti dove la velocità è cruciale e la gamma di valori da trattare è limitata, come nell'elaborazione di segnali digitali (DSP), nei sistemi embedded o nei videogiochi. L'idea di base è semplice: un numero a punto fisso non è altro che un intero, scritto in binario, a cui **decidiamo a priori** dove collocare la "virgola" (in questo caso il punto binario). Tutte le operazioni avvengono come se si trattasse di interi; solo l'interpretazione dei bit cambia.

Dalla notazione binaria alla "binary point"

Un numero binario puro, ad esempio 110101_2, rappresenta:

$$1\cdot2^5+1\cdot2^4+0\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0 = 53_{10}.$$

Se vogliamo esprimere 26,5 in binario, basta "inserire" un punto binario: 11010.1_2, che equivale a:

$$1\cdot2^4+1\cdot2^3+0\cdot2^2+1\cdot2^1+0\cdot2^0+1\cdot2^{-1} =$$

$$16+8+2+0.5=$$

$$26.5.$$

L'unica differenza fra 53 e 26,5 è la posizione del punto. Spostarlo a sinistra di una posizione significa dividere per 2; spostarlo a destra moltiplica per 2. Questa osservazione è la chiave della rappresentazione a punto fisso: **definire a priori dove si trova il punto**.

Definizione formale del tipo fixed point

Un tipo a punto fisso è descritto da due parametri:

- **larghezza** totale in bit w,
- **posizione del punto** b, contata a partire dal bit meno significativo.

Si scrive spesso fixed<w,b>. Ad esempio fixed<8,3> indica un numero di 8 bit di cui gli ultimi 3 rappresentano la parte frazionaria. Il bit pattern 00010110_2 in questo formato equivale a

$$00010.110_2 \quad = \quad 2+0.5+0.25 \quad = \quad 2.75.$$

La stessa sequenza di bit può assumere significati diversi se il punto è in posizione diversa: con fixed<8,5> diventa

$$000.10110_2 \quad = \quad 0.5+0.125+0.0625 \quad = \quad 0.6875.$$

Numeri negativi e complemento a due

Per gestire i numeri negativi si usa il **complemento a due**, esattamente come per gli interi. La bellezza del complemento a due è che somma, sottrazione e persino gli shift a sinistra o destra funzionano senza distinzioni tra positivi e negativi. Se il punto binario è fissato, spostare i bit di una posizione verso destra equivale a dividere per 2 anche per i numeri negativi, purché si effettui l'estensione del segno. Di conseguenza possiamo eseguire aritmetica su numeri a punto fisso riutilizzando l'hardware degli interi, senza circuiti speciali.

IN C++

C++ non ha un tipo “fixed point” già pronto.
Ma possiamo crearne uno **avvolgendo** un intero (per esempio `int`) e ricordandoci che la virgola è fissata in una posizione decisa da noi.
Quando facciamo calcoli, lavoriamo come se fossero semplici interi.
Quando dobbiamo **mostrare** il numero, dividiamo di nuovo per la scala corretta per far riapparire la virgola.

Un esempio passo per passo:

Diciamo che vogliamo 16 bit per la parte dopo la virgola (precisione molto alta).
Allora il fattore di scala sarà $2^{16} = 65536$.

- Per salvare 2,75: faccio 2.75 * 65536 ≈ 180224.
  Memorizzo l’intero 180224.
- Per leggere: divido 180224 / 65536 = 2.75.

EX00 MODULE CPP02

Orthodox Canonical Form

In C++ questo termine indica la cosiddetta *rule of three* (o, nelle versioni più moderne, *rule of five*).
Quando una classe **gestisce una risorsa** – qui la risorsa è il dato “raw” che rappresenta il numero a punto fisso – il compilatore genera automaticamente costruttore di copia, operatore di assegnazione e distruttore.
Ma per avere **controllo totale e comportamento esplicito**, la forma canonica ortodossa richiede che tu li scriva **tutti e quattro**:

1. **Costruttore di default** – per creare l’oggetto in uno stato valido anche senza parametri.
2. **Costruttore di copia** – per creare un nuovo oggetto partendo da un altro (copia profonda o superficiale a tua scelta).
3. **Operatore di assegnazione (=)** – per riassegnare a un oggetto già esistente il contenuto di un altro.
4. **Distruttore** – per rilasciare eventuali risorse.

Obiettivo

Realizzare una **classe C++** che rappresenti un numero **a punto fisso** (fixed-point) seguendo la **Orthodox Canonical Form**. Lo scopo: esercitarsi con la *rule of three* (costruttore di default, costruttore di copia, operatore di assegnazione e distruttore); comprendere la differenza tra **rappresentazione interna** (un intero che codifica la parte frazionaria) e **valore reale**; sperimentare l’incapsulamento: l’utente della classe manipola solo metodi pubblici, non i dettagli di memorizzazione.

Rappresentazione fixed-point

La classe adotta 8 bit fissi per la parte frazionaria. Se raw è l’intero memorizzato, il numero reale è:

valore = raw / 2^8

- **Risoluzione**: il passo minimo è $2^{-8} \approx 0.0039$.
- **Range**: con un int a 32 bit il valore massimo rappresentabile è circa $(2^{31} - 1)/256$.
- **Aritmetica**: somme e sottrazioni sono identiche a quelle tra interi; per moltiplicare o dividere occorre gestire il fattore di scala.

Queste proprietà mostrano la differenza concettuale rispetto alla floating-point IEEE 754: il fixed-point è più veloce e deterministico ma ha precisione e intervallo limitati.

Svolgimento

1. **Definizione dei membri privati**: un intero _raw per il valore e la costante static const int _fractionalBits = 8 condivisa da tutte le istanze.

2. **Implementazione dei quattro metodi canonici**:
    - default constructor che pone _raw a 0;
    - costruttore di copia che inizializza _raw con quello dell'oggetto sorgente;
    - operatore di assegnazione che copia _raw proteggendo dal self-assignment;
    - distruttore (qui vuoto, ma esplicito).
3. **Metodi di accesso**: getRawBits() restituisce l'intero, setRawBits() lo imposta.

EX01MODULE02

Obiettivo

Il passo avanti rispetto a ex00 è trasformare Fixed da "contenitore di 0" a **rappresentazione utile di numeri reali in punto fisso** con 8 bit frazionari. In pratica, la classe deve saper costruirsi da un intero o da un float, salvare internamente un **raw** intero scalato, riconvertire a float o int quando richiesto e stampare in modo naturale tramite operator<<. Rimane la **Orthodox Canonical Form** (costruttore di default, di copia, assegnazione, distruttore) per mantenere esplicito il ciclo di vita dell'oggetto.

Implicazioni teoriche

Con 8 bit di frazione stai dicendo che l'unità minima è $2^{-8} = 1/256$.

x = fixedPointValue / 2^8

Quando costruisci da `int n`, moltiplichi per 2^8 senza perdita di informazione (shift a sinistra di 8: `n << 8`). Quando costruisci da `float f`, devi **quantizzare** su passi di 1/256: salvi

raw = roundf ($f \cdot 2^8$) così minimizzi l'errore (round-to-nearest) invece di troncare.

La riconversione segue il percorso inverso: `toFloat()` divide per 28 e `toInt()` fa lo shift a destra di 8, che equivale a prendere solo la parte intera nella scala del fixed-point. Questo modello chiarisce anche i limiti: **risoluzione** di circa 0,00390625 e **range** legato all'ampiezza dell'`int` usato per il raw. Inoltre, l'overload di `operator<<` deve esprimere il significato matematico, non la rappresentazione: quindi stampa la **forma floating** (`toFloat()`), non il raw. . L'overload di `operator<<` si limita a demandare a `toFloat()`, in modo che la stampa rifletta il valore numerico e non l'encoding.

Questo esercizio consolida due idee. La prima è la **modellazione numerica**: il fixed-point è un compromesso preciso tra velocità e precisione, con una scala fissa che rende prevedibili gli errori (quantizzazione a passi di 1/256) e semplici le operazioni base (shift e somme sugli interi). La seconda è la **progettazione orientata agli oggetti**: la Orthodox Canonical Form ti costringe a ragionare sul ciclo di vita, mentre l'interfaccia mantiene chiara la distinzione tra ciò che l'utente deve vedere (un numero) e come lo conservi (un intero scalato).

Dentro l'oggetto tieni un unico intero `_fixedPointValue`. L'interpretazione matematica è fissa e indipendente da come è nato quel valore: il numero reale rappresentato è sempre

$$x = \frac{\_fixedPointValue}{2^8} = \frac{\_fixedPointValue}{256}.$$

- **Costruttore da** `int` : fa lo shift a sinistra di 8 (`raw = n << 8`). È un'operazione esatta: stai moltiplicando l'intero per 2^8 senza introdurre errore. Se `n=10` , salvi 2560 e "vedi" 10.0 quando dividi per 256.
- **Costruttore da** `float` : fa la quantizzazione su griglia di passo $1/256$:

$$\text{raw} = \text{roundf}\left(f \cdot 256\right).$$

-