13. Tipi enumerati e reali

Andrea Marongiu

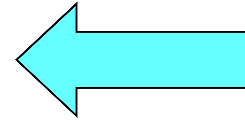
(andrea.marongiu@unimore.it)

Paolo Valente

UNIMORE

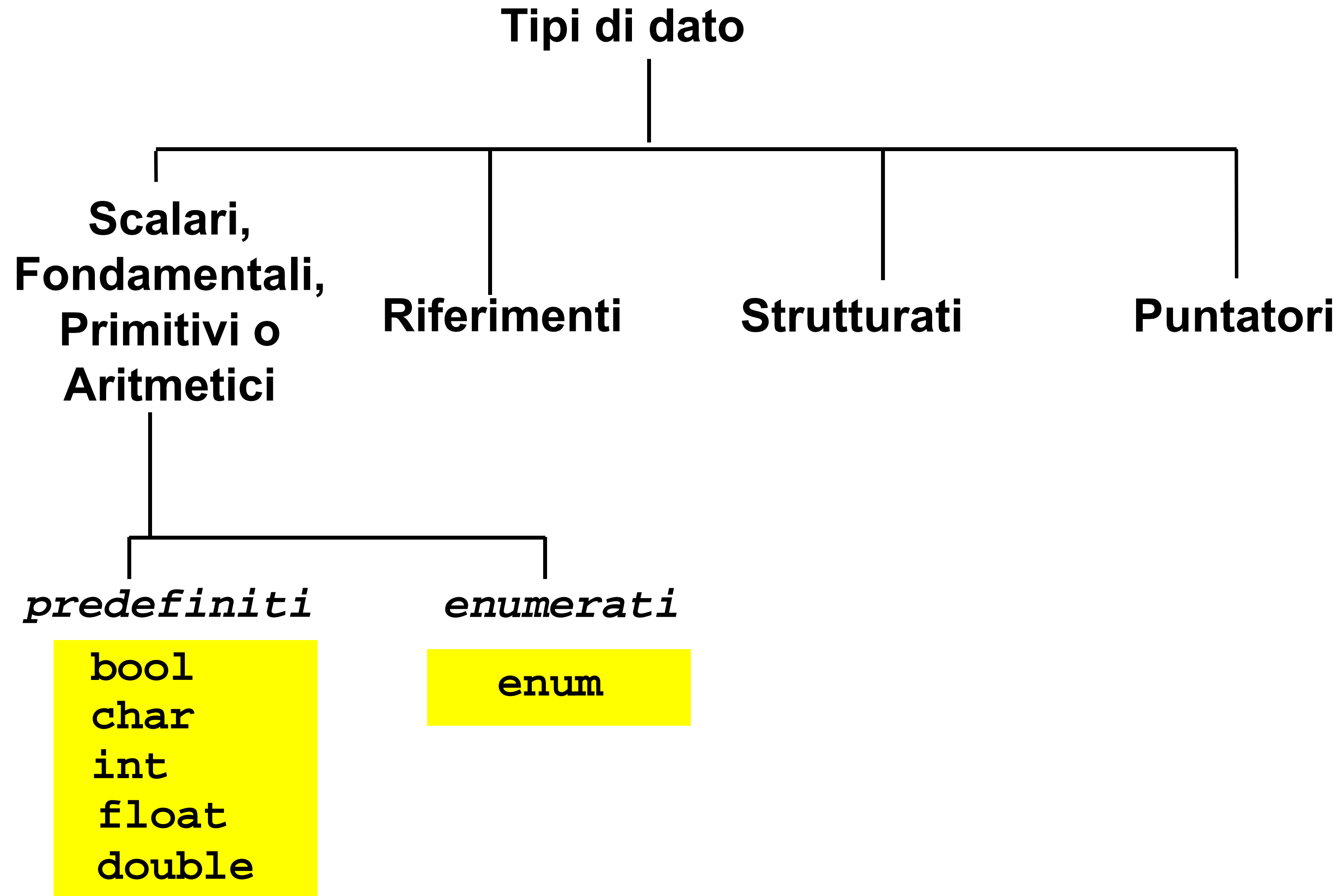
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

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 di dato



Analisi funzione

```
void fun(int i)
{
 if (i == 3)
      cout<<"Turno: mattino e pomeriggio";
 else
      cout<<"Turno: solo mattino";
}
```

- E' facile capire il senso o lo scopo di questa funzione?
- Il tipo del parametro formale ci aiuta a capire il senso della funzione?

Proposta

- No, il tipo è troppo **generico**
- Supponiamo invece che esista un tipo di dato chiamato **`giorno_lavorativo`**
  - I cui unici valori possibili sono le costanti:
  - **`lunedi martedi mercoledi giovedi venerdi`**
- E supponiamo di riscrivere la funzione utilizzando tale tipo di dato

Nuova versione

```
void fun(giorno_settimana i)
{
 if (i == giovedi)
     cout<<"Turno: mattino e pomeriggio";
 else
     cout<<"Turno: solo mattino";
}
```

- Adesso è immediato capire che la funzione serve a stampare dei turni di lavoro in base al giorno della settimana!
- Questo non è l'unico vantaggio del disporre del nuovo tipo di dato che stiamo 'inventando'

Domanda

- Quando, nella precedente funzione, la condizione nell'`if` è falsa, cosa sappiamo di certo sul valore del parametro formale `i`?

Risposta e nuova domanda

- Che i possibili valori della variabile sono **solo** i giorni della settimana diversi da giovedì
- Non è quindi possibile, per errore, passare un valore del parametro formale che non sia uno dei giorni lavorativi
- Abbiamo la stessa certezza nel caso in cui **i** sia di tipo **`int`**?

Risposta

- No
- Se `i != 3`, non abbiamo nessuna garanzia che il suo valore sia correttamente uguale al valore di uno degli altri giorni della settimana
  - Il valore di **i** potrebbe essere troppo grande o perfino negativo!
- I due problemi di leggibilità e correttezza appena visti sono alla base dell'introduzione del tipo enumerato …

Tipo enumerato 1/2

- **Insieme di costanti** intere definito dal programmatore
  - ciascuna individuata da un identificatore (nome) e detta **enumeratore**
- Esempio di dichiarazione:
-
- **`enum colori_t {rosso, verde, giallo} ;`**
  - dichiara un tipo enumerato di nome **`colori_t`** e tre costanti intere (enumeratori) di nome **`rosso`**, **`verde`** e **`giallo`**
  - gli oggetti di tipo **`colori_t`** potranno assumere come valori solo quelli dei tre enumeratori
  - agli enumeratori sono assegnati numeri interi consecutivi a partire da zero, a meno di inizializzazioni esplicite (che vedremo fra poco)

Tipo enumerato 2/2

- Rimanendo sull'esempio della precedente slide
  - mediante il tipo **`colori_t`** sarà possibile definire nuovi oggetti mediante delle definizioni, con la stessa sintassi usata per i tipi predefiniti
  - Così come si può scrivere
  - **`int a ;`**
  - si potrà anche scrivere
  - **`colori_t a ;`**
    - il cui significato è quello di definire un oggetto di nome **`a`** e di tipo **`colori_t`**
  - I valori possibili di oggetti di tipo **`colori_t`** saranno quelli delle costanti **`rosso`**, **`verde`** e **`giallo`**
  - Quindi l'oggetto **`a`** definito sopra potrà assumere solo i valori **`rosso`**, **`verde`** e **`giallo`**

Sintassi

- Dichiarazione di un tipo enumerato:
-

<*dichiarazione_tipo_enumerato*> ::=
  enum <*identificatore*> {<*lista_dich_enumeratori*>} **;**

<*lista_dich_enumeratori*> ::=
  <*dich_enumeratore*> **{, <*dich_enumeratore*>}**

<*dich_enumeratore*> ::=
  <*identificatore*> **[= <*espressione*>]**

Ripetuto zero o più volte

Ripetuto zero o una volta

Inizializzazione e visibilità

- Come già detto agli enumeratori sono associati per default valori interi consecutivi a partire da 0
- Esempio: gli enumeratori del precedente tipo **`colori_t`** valgono 0 (**`rosso`**), 1 (**`verde`**) e 2 (**`giallo`**)
- La dichiarazione dell'identificatore di un tipo enumerato segue le stesse regole di visibilità di una generica dichiarazione
- Nel campo di visibilità dell'identificatore di un tipo enumerato
  - si possono utilizzare i suoi enumeratori
  - si può utilizzare il nome del tipo per definire variabili di quel tipo
  - Esempio:
  - **`colori_t c ;`**
  - **`colori_t d = rosso ;`**

Note sui tipi enumerati 1/2

- Attenzione, se si dichiara una variabile o un nuovo enumeratore con lo stesso nome di un enumeratore già dichiarato, da quel punto in poi si perde la visibilità del precedente enumeratore.

Esempio:

```
enum Giorni {lu, ma, me, gi, ve, sa, do} ;
enum PrimiGiorni {do, lu, ma, gi} ;
// da qui in poi non si vedono più gli enumeratori
// lu, ma, gi e do del tipo Giorni
```

- Un tipo enumerato è totalmente ordinato. Su un dato di tipo enumerato sono applicabili tutti gli operatori relazionali. Continuando i precedenti esempi:
  - `lu < ma` → vero
  - `lu >= sa` → falso
  - `rosso < giallo` → vero

Note sui tipi enumerati 2/2

- Se si vuole, si possono inizializzare a piacimento le costanti:

```
enum Mesi {gen=1, feb, mar, ... }   ;
     // Implica: gen = 1, feb = 2, mar = 3, ...
enum romani { i=1, v = 5, x = 10, c = 100 }   ;
```

- E' possibile definire direttamente una variabile di tipo enumerato, senza dichiarare il tipo a parte
- <*definizione_variabile_enumerato*> ::=
- **enum** { <*lista_dich_enumeratori*> } <*identificatore*> ;
  - Esempio: `enum {rosso, verde, giallo} colore ;`
  - Nel campo di visibilità della variabile è possibile utilizzare sia la variabile che gli enumeratori dichiarati nella sua definizione

Occupazione di memoria

- Lo spazio esatto occupato in memoria da un oggetto di tipo enumerato dipende dal compilatore
  - Tipicamente: stessa occupazione di memoria (in numero di byte) del tipo **`int`**
- Per un dato tipo enumerato, l'insieme di valori possibili è però ovviamente limitato ai suoi soli enumeratori
- Se un dato programma per funzionare correttamente ha bisogno che gli enumerati occupino un determinato spazio in memoria
  - Tale programma funziona solo se il compilatore con cui è compilato rispetta tale assunzione
  - Il programma non è quindi portabile

Controllo nelle operazioni 1/2

- Se non si effettuano mai operazioni tra enumerati ed oggetti di altro tipo (ad esempio interi), non si corrono i seguenti rischi
  - un oggetto di tipo enumerato contiene un valore diverso da uno dei suoi enumeratori
  - un programma fa affidamento sul valore esatto di qualche enumeratore, e quindi non è più corretto se tale valore cambia
- Inoltre il compilatore aiuta il programmatore a non commettere l'errore di assegnare valori impropri ad un oggetto di tipo enumerato
  - Infatti proibisce di assegnare ad un oggetto di tipo enumerato un valore di tipo diverso dal tipo dell'oggetto enumerato stesso
  - Ad esempio, l'istruzione
  - **`colore_t c = 100;`**
  - causa un errore a tempo di compilazione

Controllo nelle operazioni 2/2

- Però sono lecite operazioni pericolose tipo:

```
colore_t c = static_cast<colore_t>(100);

if (rosso == 1) cout<<"Uguale ad 1"<<endl;

enum soprannome_t {tizio, caio};
if (caio < verde) cout<<"caio < verde"<<endl;
```

- Il fatto che tali operazioni siano legali viola la tipizzazione forte che si cerca di garantire nel linguaggio C++
- Questo problema è affrontato nello standard C++11 nel modo seguente

enum class in C++11 1/3

- A partire dallo standard C++11, è stato introdotto un nuovo tipo di dato, denotato come **`enum class`**
- La sintassi della dichiarazione di un nuovo tipo
- **`enum class`** è la seguente
-
- *<dichiarazione_tipo_enumeration>* ::=
- **`enum class`** *<identificatore>* **`{`** *<lista_dich_enumeratori>* **`} ;`**
- Identica alla dichiarazione di un nuovo tipo **`enum`**, a parte l'aggiunta della parola chiave **`class`**

enum class in C++11 2/3

- La sintassi della definizione di oggetti di tipo
- **`enum class`** è identica a quella della definizione di oggetti di tipo **`enum`**
- Esempio
- **`enum class colore2_t {blu, nero, bianco};`**
- **`colore2_t col;`**
- A differenza del tipo **`enum`**, per utilizzare un enumeratore di un dato tipo **`enum class`**, bisogna aggiungere come prefisso il nome del tipo seguito da **`::`**
  - Esempi (data la dichiarazione nel precedente esempio)
  - **`cout<<blu; // ERRATO`**
  - **`cout<<colore2_t::blu; // CORRETTO`**
  - Questo permette a due o più tipi enumerati di avere gli enumeratori con lo stesso nome senza che sorgano problemi di compilazione o ambiguità

enum class in C++11 3/3

- L'altro grande vantaggio in termini di controllo di tipo è che con i tipi **`enum class`** non è possibile alcuna delle operazioni pericolose permesse con il tipo **`enum`**
  - Non è però possibile neanche stampare un oggetto di tipo **`enum class`** passandolo semplicemente all'operatore **`<<`**
- Il tipo **`enum class`** permette infine di decidere anche esattamente il tipo di dato sottostante, ossia il tipo di dato utilizzato per memorizzare i valori degli enumeratori
  - Si può quindi di decidere anche quanta memoria viene occupata dagli oggetti di un dato tipo
  - **`enum class`**
  - Non vediamo la relativa sintassi in questo corso

Utilizzo enum class

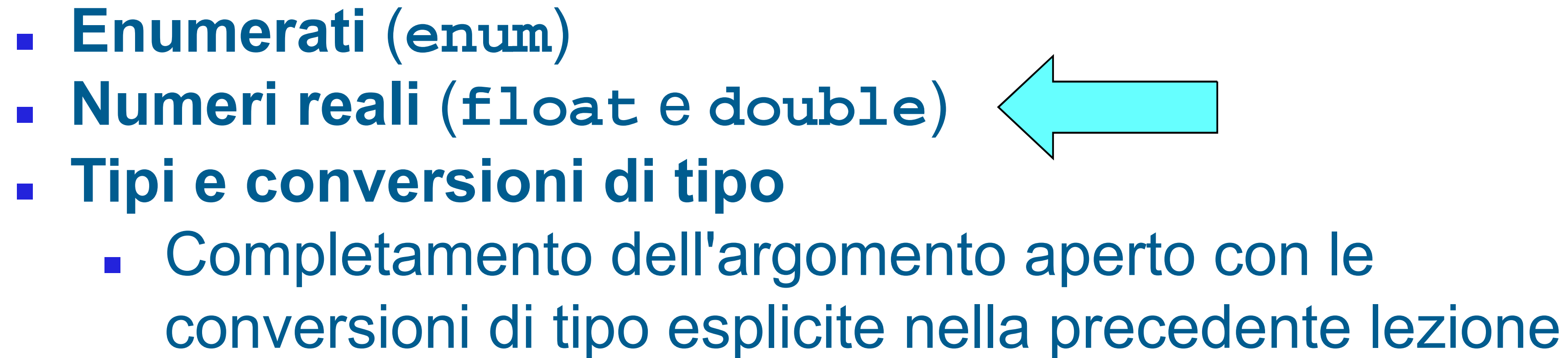
- Elemento importante da considerare per decidere se utilizzare il tipo **`enum class`** oppure no
  - Se utilizzate **`enum class`** il programma non è compilabile con i compilatori che non supportano (ancora) lo standard C++11

Benefici del tipo enumerato

- Decisamente migliore leggibilità
- Indipendenza del codice dai valori esatti e dal numero di costanti (enumeratori)
  - Conseguenze importantissime:
    - se cambio il valore di un enumeratore, non devo modificare il resto del programma
    - posso aggiungere nuovi enumeratori senza dover necessariamente modificare il resto del programma
- Maggiore robustezza agli errori
  - Se si usano solo gli enumeratori **non è praticamente possibile usare valori sbagliati**
- Quindi: impariamo da subito ad utilizzare gli enumerati e non gli interi **ovunque i primi siano più appropriati dei secondi**

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



Notazione posizionale

- Prima di discutere i numeri reali occorre fare una divagazione sulla notazione posizionale

Ci sono solo 10 tipi di persone al mondo:

- *quelle che conoscono la rappresentazione dei numeri in base 2, e*
- *quelle che non la conoscono ...*

Base 2

- Per capire fino in fondo come sono rappresentate le informazioni in un calcolatore occorre conoscere la rappresentazione dei numeri in base 2

- Il motivo è che le informazioni sono rappresentate come sequenze di bit, ossia cifre con due soli possibili valori

Basi e cifre 1/2

- Partiamo dalla rappresentazione di un numero in una generica base
- Cominciamo dalla rappresentazione dei numeri naturali

Basi e cifre 2/2

- Rappresentazione di un numero in una data base: sequenza di cifre
- *Cifra*: simbolo rappresentante un numero
- *Base*: numero (naturale) di valori possibili per ciascuna cifra
- In base $b > 0$ si utilizzano b cifre distinte, per rappresentare i valori 0, 1, 1 + 1, 1 + 1 + 1, ..., $b - 1$

Cifre e numeri in base 10

- Es: in base 10 le cifre sono

0 che rappresenta il valore 0

1 che rappresenta il valore 1

2 che rappresenta il valore 1+1

3 che rappresenta il valore 1+1+1

.

.

.

9 che rappresenta il valore 1+1+1+1+1+1+1+1+1

Simbolo grafico

Concetto astratto di numero naturale

Notazione posizionale

- Rappresentazione di un numero su n cifre in base b:

$$a_{n-1} \quad a_{n-2} \quad a_{n-3} \ldots a_1 \; a_0$$

$$a_i \; \varepsilon \; \{0, 1, \ldots, b - 1\}$$

- Es: Notazione decimale:
- $b = 10$, $a_i \; \varepsilon \; \{0, 1, 2, \ldots, 9\}$
- 345 => $a_2 = 3$, $a_1 = 4$, $a_0 = 5$

Notazione

- Per rendere esplicita la base utilizzata, si può utilizzare la notazione

$$[\, x \,]_b$$

$$a_i \ \varepsilon \ \{0, 1, \ldots, b - 1\}$$

dove x è una qualsiasi espressione, ed il cui significato è che ogni numero presente nell'espressione è rappresentato in base b

Esempi in base 10

$[\ 345\ ]_{10}$

$[\ 2 * 10 + 5 * 1\ ]_{10}$

Notazione posizionale

- $[\ a_{n-1}\ \ a_{n-2}\ \ a_{n-3}\ \ldots\ a_1\ \ a_0\ ]_b =$

$[a_0 * 1 + a_1 * b + a_2 * b^2 + a_3 * b^3 + \ldots + a_{n-1} * b^{n-1}\ ]_b =$

$[\ \sum_{i=0,..,n-1}\ a_i * b^i\ ]_b$ *Peso cifra i-esima*

- Es: $b = 10$, $a_i \, \varepsilon \, \{0, 1, 2, \ldots, 9\}$
- $[\ 345\ ]_{10} = [3 * 10^2 + 4 * 10 + 5 * 1]_{10}$

"yo cuento como un cero a la izquierda"
… io conto come uno zero a sinistra

Calcoli

- Si utilizzano degli algoritmi
- Esattamente quelli imparati alle elementari per la base 10
- Esempio: per sommare due numeri, si sommano le cifre a partire da destra e si utilizza il riporto

Notazione binaria

- Base 2; 2 cifre:
  - 0, 1
- La cifra nella posizione *i-esima* ha peso 2^i

Esempi (*configurazioni di bit*):

$[0]_{10} = [0]_2$

$[1]_{10} = [1]_2$

$[2]_{10} = [10]_2 = [1*2 + 0*1]_{10}$

$[3]_{10} = [11]_2 = [1*2 + 1*1]_{10}$

Base 16 1/2

- Una base che risulta spesso molto conveniente è la base 16
- Perché ogni cifra in base sedici corrisponde ad una delle possibili combinazioni di 4 cifre in base 2
- Quindi, data la rappresentazione in base 2 di un numero naturale, la sua rappre-sentazione in base 16 si ottiene dividendo la sequenza in base in sotto-sequenze consecutive da 4 cifre ciascuna, partendo da destra, e convertendo ciascuna sotto-sequenza di quattro cifre binarie nella corrispondente cifra in base 16

Base 16 2/2

- Viceversa, data la rappresentazione in base 16 di un numero naturale, il corrispondente numero in base 2 si ottiene convertendo semplicemente ciascuna cifra della rappresentazione in base 16 nella corrispondente sequenza di 4 cifre inbase 2

Notazione esadecimale

- Base 16, 16 cifre:
  - 0, 1, 2, ..., 9, A, B, C, D, E, F
- Valore cifre in decimale:
  - 0, 1, 2, ..., 9, 10, 11, 12, 13, 14, 15
- La cifra nella posizione *i-esima* ha peso 16^i

Esempi:

$[0]_{10} = [0]_{16}$

$[10]_{10} = [A]_{16}$

$[18]_{10} = [12]_{16} = [1*16 + 2*1]_{10}$

Rappresentazione naturali

- In una cella di memoria o in una sequenza di celle di memoria si può memorizzare con facilità un numero naturale memorizzando la configurazione di bit corrispondente alla sua rappresentazione in base 2
- Questa è la tipica modalità con cui sono memorizzati i numeri naturali
- Coincide con gli esempi che abbiamo già visto in lezioni precedenti

Rappresentazione interi 1/2

- Come rappresentare però numeri con segno?

- Non esiste un elemento all'interno delle celle, che sia destinato a memorizzare il segno

- Come potremmo cavarcela?

Rappresentazione interi 2/2

- Un'idea sarebbe quella di utilizzare uno dei bit per il segno
  - 0 per i valori positivi
  - 1 per i valori negativi

- Il problema è che sprechiamo una configurazione di bit, perché avremmo due diverse rappresentazioni per il numero 0
  - Una col segno positivo
  - Una col segno negativo

Complemento a 2

- Per ovviare a questo problema, i numeri con segno sono tipicamente rappresentati in complemento a 2

- Se n è un numero minore di 0, allora, anziché memorizzare il numero originale n, si memorizza il risultato della somma algebrica

$$2^N + n$$

- dove N è il numero di bit su cui si intende memorizzare il numero

Vantaggi

- C'è una sola rappresentazione per lo 0
- Gli algoritmi di calcolo delle operazioni di somma, sottrazione, moltiplicazione e divisione sono gli stessi dei numeri naturali rappresentati in base 2

Nota 1/2

- Senza entrare in ulteriori dettagli
  - una sequenza di *N* bit che rappresenta un numero negativo ha sempre il bit più a sinistra uguale ad 1

  - la rappresentazione in complemento a due di un numero positivo su *N* bit è uguale alla sua rappresentazione in base 2

Nota 2/2

- Quindi una configurazione di bit con il bit più a sinistra ad 1 rappresenta
  - un valore positivo se sta rappresentando un numero naturale in base 2
  - un valore negativo se sta rappresentando un numero in complemento a 2

Rappresentazione **int**

- Gli oggetti di tipo **int** sono tipicamente rappresentati in complemento a 2
- Adesso dovrebbe esservi più chiaro perché è vero che:

“Ci sono solo 10 tipi di persone al mondo: quelle che conoscono la rappresentazione dei numeri in base 2, e quelle che non la conoscono”

Numeri reali

- In C/C++ si possono utilizzare numeri con una componente frazionaria (minore dell'unità)

Ad esempio:

```
24.2          .5
```

- Tali numeri sono comunemente chiamati reali

Letterali reali

Si possono utilizzare i seguenti formati:

```
24.0                        .5
2.4e2 = 2.4*10^2            240.0e-1 = 240.0*10^-1
```

- La notazione scientifica è utile per scrivere numeri molto grandi o molto piccoli
- Per indicare che una costante letterale è da intendersi come reale anche se non ha cifre dopo la virgola, si può terminare il numero con un punto

Esempio:

```
123.
```

Operatori reali

Operatori aritmetici	**Tipo del risultato**
`+` `-` `*` `/`	`float` o `double`

Attenzione: la divisione è quella reale

Operatori relazionali

`==` `!=`	`bool` (`int` in C)
`<` `>` `<=` `>=`	`bool` (`int` in C)

Esempi

5. / 2. == 2.5
2.1 / 2. == 1.05
7.1 > 4.55 == `true`, oppure 1 in C

Stampa numeri reali 1/2

- Come sappiamo, quando si inserisce un numero di tipo **`int`** sull'oggetto *cout* mediante l'operatore <<, viene immessa sullo *stdout* la sequenza di caratteri e cifre che rappresenta quel numero
  - Lo stesso vale per i numeri reali

- L'esatta sequenza di caratteri dipenderà da come è configurato l'oggetto *cout* (vedremo meglio in seguito)
  - Ad esempio, nella configurazione di default dell'oggetto di *cout*, la seguente riga di codice
  - **`cout<<-135.3 ;`**
  - immette sullo *stdout* la sequenza di caratteri:
  - **`-135.3`**

Stampa numeri reali 2/2

- In particolare, stampa solo un numero di cifre dopo la virgola ragionevole
- Il numero stampato può quindi non coincidere col numero in memoria

Numeri reali

- Come ogni altro tipo di dato (interi, booleani, caratteri, enumerati), anche i numeri reali sono memorizzati sotto forma di sequenze di bit
  - Più in particolare, così come un numero di tipo **int**, un numero reale è memorizzato in una sequenza di celle di memoria contigue

- Quante celle di memoria sono utilizzate e quali configurazioni di bit sono memorizzate in tali celle dipende dallo schema con cui il numero è rappresentato in memoria e dalla precisione desiderata

- Come stiamo per vedere nelle seguenti slide ...

Rappresentazioni numeri reali

- Esistono tipicamente due modi per rappresentare un numero reale in un elaboratore:
  - **Virgola fissa:** Numero massimo di cifre intere e decimali deciso a priori
    - Esempio: se si utilizzano 3 cifre per la parte intera e 2 per la parte decimale, si potrebbero rappresentare i numeri:
    - 213.78 184.3 4.21
    - ma non
    - 2137.8 3.423 213.2981
  - **Virgola mobile:** Numero massimo totale di cifre, intere e decimali, deciso a priori, ma posizione della virgola libera
    - Esempio: se si utilizzano 5 cifre in totale, si potrebbero rappresentare tutti i numeri del precedente esempio in virgola fissa, ma anche
    - 213.78 2137.8 .32412 12617.
    - ma non
    - .987276 123.456 1.321445

Componenti virgola mobile

- Si decide a priori il numero massimo di cifre perché questo permette una rappresentazione abbastanza semplice dei numeri in memoria, nonché operazioni più veloci
- Un numero reale è rappresentato (e quindi memorizzato) di norma mediante tre componenti:
  - **Segno**
  - **Mantissa** (*significand*), ossia le cifre del numero
  - **Esponente** in base 10:
- A parte il segno, il numero si immagina nella forma

$$\textit{mantissa} * 10^{\textit{esponente}}$$

  - Tipicamente la mantissa è immaginata come un numero a virgola fissa, con la virgola posizionata sempre subito prima (o in altre rappresentazioni subito dopo) della prima cifra diversa da zero

Calcolo rappresentazione 1/2

- La mantissa di un numero reale si ottiene semplicemente spostando la posizione della virgola del numero di partenza
- Partiamo per esempio dal numero 12.3
  - La virgola si trova subito dopo la seconda cifra
  - Per arrivare da questo numero ad una mantissa che abbia la virgola subito prima della prima cifra, spostiamo la virgola di due posizioni verso sinistra
    - Otteniamo .123
  - Per ottenere infine la rappresentazione di 12.3 nella forma

  $mantissa * 10^{esponente}$, ossia nella forma $.123 * 10^{esponente}$

  dobbiamo trovare il valore corretto dell'esponente
    - Tale valore è uguale al numero di posizioni di cui abbiamo spostato la virgola, ossia $12.3 = .123 * 10^2$

Calcolo rappresentazione 2/2

- In generale,
  - Se la mantissa è ottenuta spostando la virgola di *n* posizioni **verso sinistra**, allora l'esponente è uguale ad *n*
    - Come nel precedente esempio
  - Se la mantissa è ottenuta spostando la virgola di n posizioni **verso destra**, allora l'esponente è uguale a *-n*
    - Ad esempio, la mantissa di .0123 è .123, ottenuta spostando la virgola di una posizione verso destra, e la rappresentazione del numero è quindi $.123 * 10^{-1}$

Esempi

- La notazione scientifica, già vista nell precedenti slide, torna utile per evidenziare le precedenti componenti nella rappresentazione di un numero reale:
- *mantissa***e***esponente = mantissa*$*10^{esponente}$
- Esempi:

Numero	Notazione Scientifica	Segno	Mantissa	Esponente
123	.123e3	+	.123	3
0.0123	.123e-1	+	.123	-1
0.123	.123e0	+	.123	0
-1.23	-.123e1	-	.123	1

Domanda

- Perché memorizzare nella mantissa solo numeri con la prima cifra dopo la virgola diversa da zero?

Risposta

- Per non sprecare bit per memorizzare tali cifre a 0
- Si riesce comunque a riottenere il numero originale giocando opportunamente con l'esponente

Tipi float e double

- Nel linguaggio C/C++ i numeri reali sono rappresentati mediante i tipi **`float`** e **`double`**
  - Sono numeri in virgola mobile
  - Mirano a rappresentare (con diversa precisione) **un sottoinsieme** dei numeri reali
  - I tipi **`float`** e **`double`** (così come **`int`** per gli interi), sono solo un'approssimazione dei numeri reali, sia come
    - **precisione**, ossia numero di cifre della mantissa
      - torneremo più in dettaglio sul concetto di precisione a breve
    - sia come **intervallo** di valori rappresentabili

IEEE 754

- I numeri **`float`** e **`double`** sono tipicamente rappresentati/memorizzati in conformità allo standard
- IEEE 754
  - Fondamentalmente, sia la mantissa che l'esponente sono memorizzati in base 2 e non in base 10

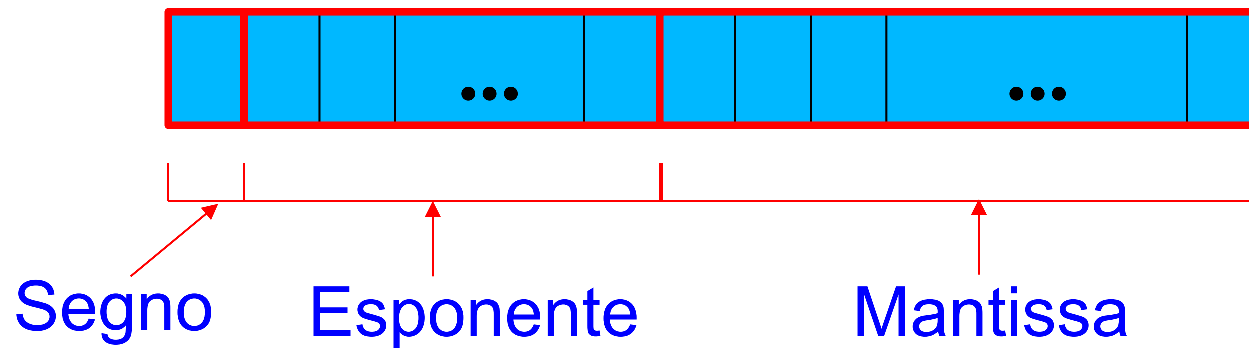
Quindi, un numero **`float`** o **`double`** è di fatto rappresentato in memoria nella forma

$\textit{mantissa} * 2^{\textit{esponente}}$

- In particolare: ...

Rappresentazione in memoria

- Un numero `float` o `double` è memorizzato come una sequenza di bit:



- Tale sequenza di bit occupa tipicamente più celle contigue in memoria

Domanda

- Come si potrebbero rappresentare esponenti di valore negativo nella precedente rappresentazione dei numeri reali?

Offset

- Si potrebbe adottare il complemento a 2
- Non è questa la soluzione effettivamente adottata
- Invece di memorizzare il valore effettivo dell'esponente *exp*
  - Si memorizza il risultato delle seguente somma
  - *exp* + *offset*
  - *offset* è un numero intero predefinito
- Quindi, dato il numero *num* memorizzato nel campo esponente
  - L'esponente effettivo è il risultato della seguente sottrazione
  - *num - offset*

Dettaglio sulla mantissa

- Anche la mantissa differisce leggermente da quella illustrata finora
- Si utilizza una tecnica che permette di risparmiare un ulteriore bit
  - Siccome, in base 2, la prima cifra della mantissa, dovendo essere diversa da 0, è uguale ad 1
  - Allora tale cifra non si memorizza affatto
  - Si memorizzano solo le cifre successive
- Infine, invece di memorizzare un numero che si assume avere la virgola subito prima della prima cifra
  - Si assume che la virgola sia subito dopo la precedente cifra uguale ad 1
- Quindi, in base 2, la mantissa ha la forma 1.xxxxxxxx

Precisione

- Definiamo **precisione** *P* di un tipo di dato numerico in una data base *b* come il numero massimo di cifre in base *b* tali che qualsiasi numero rappresentato da *P* cifre appartiene a tale tipo di dato
  - Indipendentemente da dove si colloca la virgola in tale rappresentazione

- **Esempi**
  - un tipo di dato che possa contenere numeri interi da 0 a 9999, ha una precisione in base 10 uguale a 4
    - Ossia di 4 cifre decimali
  - un tipo di dato che possa contenere numeri in virgola fissa da 0.00 a 9.99 ha una precisione in base 10 uguale a 3 (ossia di tre cifre decimali)

Valori tipici per float a double

(non necessariamente validi per tutte le architetture)

Tipo	Precisione	Intervallo di valori assoluti
`float`	6 cifre decimali	$3.4*10^{-38}$ ... $3.4*10^{38}$
`double`	15 cifre decimali	$1.7*10^{-308}$ ... $1.7*10^{308}$

Occupazione di memoria:

`float`	4 byte
`double`	8 byte
`long double`	10 byte

Domanda

- Si possono rappresentare TUTTI i numeri reali inclusi negli intervalli riportati per i **double** ed i **float** nella precedente slide?

Risposta

No

- A causa della precisione limitata vi sono numeri reali che, pur ricadendo in tali intervalli, non sono rappresentabili con un **double** o un **float**

Esempio

- Siccome la precisione di un **float** è di sole 6 cifre in base 10, allora si può rappresentare il numero

141231 che, **ipotizzando per semplicità rappresentazione in base 10**, sarebbe memorizzato come .141234e6

- ma non il numero 1412313

Domanda

- Come si riescono allora a rappresentare, con il tipo **double** o **float**, numeri con un numero di cifre più grande della precisione di cui si dispone?

Risposta

- Sfruttando l'esponente
- Ad esempio, il tipo **float** permette di rappresentare numeri con 38 cifre decimali dopo lo zero
- Ma solo le prime 6 cifre decimali possono essere l'una diversa dall'altra
- **Ipotizzando per semplicità rappresentazione in base 10**, le cifre restanti possono essere solo un gran numero di zeri, che si possono aggiungere assegnando un valore molto elevato all'esponente

Esempio:

- **ipotizzando per semplicità rappresentazione in base 10**, in un numero di tipo **float** si potrebbe memorizzare
- 121323000000000000 nella forma .121323e18
- ma non
- 121323231000000000

Numero di cifre e precisione

- Attenzione quindi a non confondere l'alto numero di cifre che può avere un numero di tipo **float** o **double**, con il numero di cifre che determinano la precisione di tali tipi di dato

Domanda

- Da cosa è determinata la precisione del tipo **float** o **double** in una qualsiasi base?
- In particolare, a cosa è uguale la precisione in base 2 del tipo **float** e del tipo **double**?

Precisione reali

- Dal numero di cifre della mantissa
- In particolare, la precisione in base 2 è uguale al numero di cifre della mantissa

Domanda

- Qual è la precisione in base 2 del tipo **int** supponendo che sia memorizzato in complemento a 2 su 32 bit?

Precisione in base 2 degli interi

- 31
- Uno dei 32 bit, quello più significativo, è utilizzato in pratica per determinare il segno del numero
- Sono i restanti 31 bit che in sostanza si usano per le cifre sia dei numeri positivi che dei numeri negativi rappresentabili
- In generale, la precisione di un tipo intero i cui valori sono rappresentati in complemento a due è uguale al numero di cifre binarie con cui sono rappresentati tali valori, meno uno

Conversione da reale ad intero

- La conversione da reale a intero è tipicamente effettuata per *troncamento*
  - Si conserva cioè solo la parte intera del numero di partenza
- Il valore convertito dovrà appartenere a qualcuno dei tipi numerabili (int, char ed altri che vedremo)
  - Se il numero di partenza è troppo grande, si verifica un *overflow* all'atto della conversione verso uno di tali tipi integrali
  - Torneremo su questo ed altri problemi legate alle conversioni tra reali ed interi (e viceversa) nelle prossime slide

Problemi di rappresentazione 1

- Siccome il numero di cifre utilizzate per rappresentare un numero reale è limitato, si possono verificare approssimazioni (*troncamenti*) nella rappresentazione di un numero reale con molte cifre
- Esempio: Il numero 290.00124
  - se si avessero massimo 6 cifre diverse a disposizione (come col tipo **float**) potrebbe essere rappresentato come .290001e+3
  - Tuttavia, questa rappresentazione trasformerebbe il numero originario
  - 290.00124 → 290.001
  - In molte applicazioni questa approssimazione non costituisce un problema, ma in altre applicazioni, come ad esempio quelle di calcolo scientifico, costituisce una **seria fonte di errori**

Problemi di rappresentazione 2

- Il numero di cifre limitato non è l'unica fonte di problemi di rappresentazione

- Ad esempio, come si può rappresentare 0.1 nella forma *mantissa* * $2^{esponente}$ con la mantissa rappresentata in base 2?
  - Bisogna trovare una coppia mantissa/esponente opportuna

- In merito, consideriamo che si possono rappresentare numeri minori di 1 in base 2 utilizzando la notazione a punto così come si fa per la base 10

  Ad esempio:

  - $[0.1]_2 = [0 + 1*2^{-1}]_{10} = 0.5$ $[0.01]_2 = [0 + 0*2^{-1} + 1 * 2^{-2}]_{10}$

  - Ma $[0.1]_{10} = [10^{-1}]_{10} = [???]_2$

Risposta

- Ogni numero frazionario, ossia minore dell'unità, che sia rappresentato da una qualsiasi sequenza di cifre dopo la virgola in base 2, è uguale alla somma di numeri razionali con una potenza di 2 al denominatore (uno per ogni cifra)
  - In totale è quindi uguale ad un numero razionale con una potenza di 2 al denominatore
- Quindi solo i numeri razionali frazionari che hanno una potenza di 2 al denominatore si possono esprimere con una sequenza finita di cifre binarie
- $[0.1]_{10}$ non si può scrivere come un numero razionale con una potenza di 2 al denominatore
- Quindi **non esiste nessuna rappresentazione finita in base 2** di $[0.1]_{10}$
  - Tale numero sarà pertanto **necessariamente memorizzato in modo approssimato**

Operazioni tra reali ed interi

- Se si esegue una operazione tra un oggetto di tipo **int**, **enum** o **char** ed un oggetto di tipo reale, si effettua di fatto la variante reale dell'operazione
  - In particolare, nel caso della divisione, si effettua la divisione reale
- Vedremo in seguito il motivo …
- Svolgere a casa l'esercizio *divis_reale2.cc*

Domanda

- Come è rappresentato il valore 0 in un float o in un double?

Risposta

- In nessun modo
  - La mantissa, per definizione, ha la prima cifra **diversa** da 0
  - Quindi la mantissa non potrà **mai** essere uguale a 0
- Lo 0 è rappresentato quindi in modo approssimato
  - Mediante il numero più piccolo rappresentabile

Confronto approssimato

- Ovviamente possono verificarsi errori dovuti al troncamento o all’arrotondamento di alcune cifre decimali anche nell'esecuzione delle operazioni
- In generale, meglio evitare l’uso dell’operatore **`==`**
  - I test di uguaglianza tra valori reali (in teoria uguali) potrebbero non essere verificati
  - Ad esempio, non sempre vale:
  - **`(x / y) * y  ==  x`**
- Meglio utilizzare "un margine accettabile di errore":
  - **`x == y  →  (x <= y+epsilon) && (x >= y-epsilon)`**
  - dove, ad esempio,
  - **`const double epsilon = 1e-7 ;`**
- Quale margine scegliere?
  - Dipende dal problema che si sta risolvendo

Riassunto errori comuni

- Confusione tra divisione fra interi e divisione fra reali
  - Stesso simbolo /, ma differente significato
- Tentativo di uso dell’operazione di modulo (%) con numeri reali, per i quali non è definita
- Uso erroneo dell'operatore di assegnamento (=) al posto dell'operatore di uguaglianza (==)