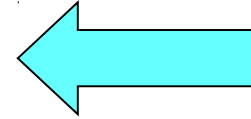


Lezione 10

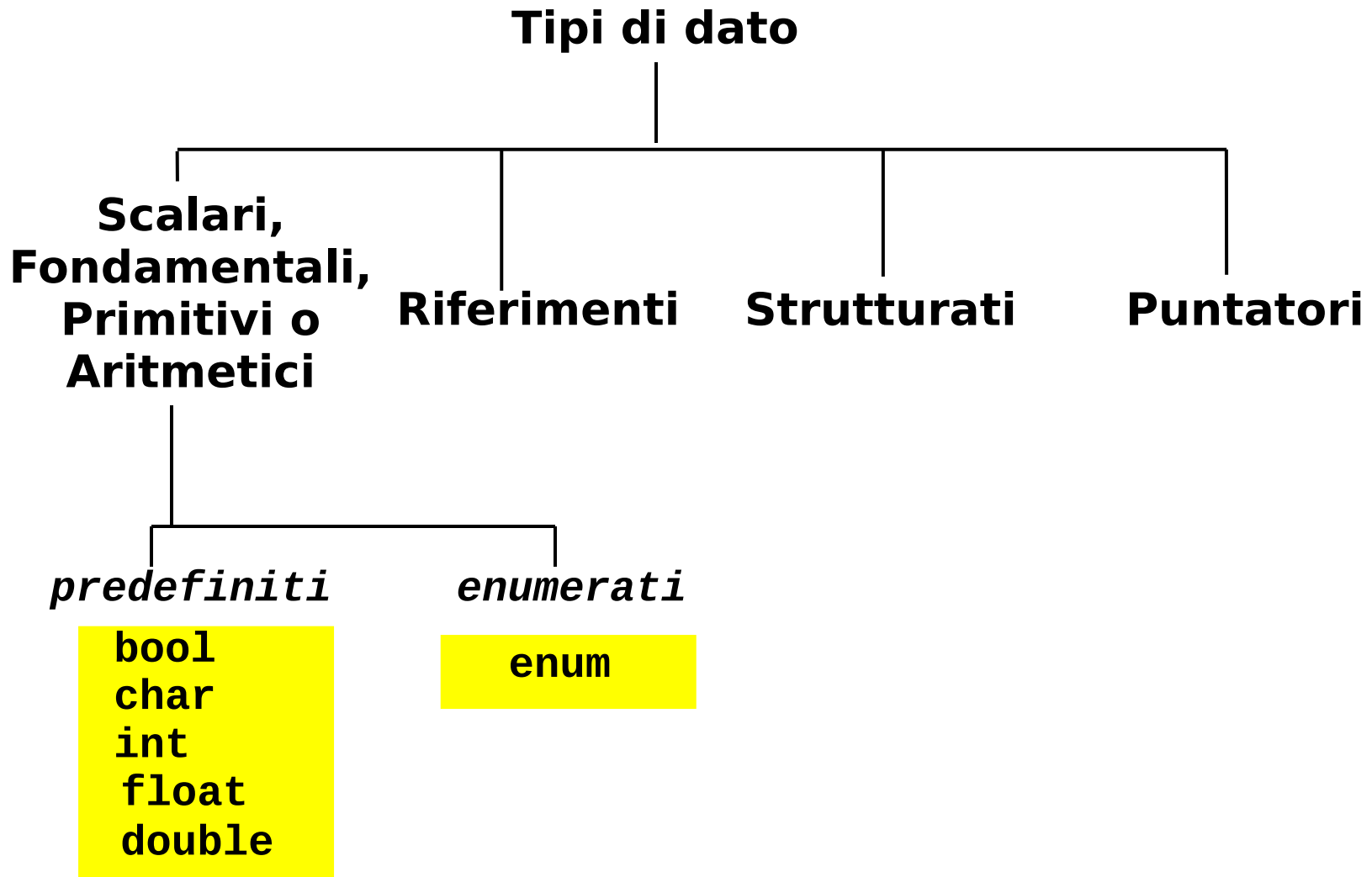
Tipo enum
Tipi float e double
Tipi e conversioni di tipo

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?

- 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:
lunedì martedì mercoledì giovedì venerdì
- E supponiamo di riscrivere la funzione utilizzando tale tipo di dato

Nuova versione

```
void fun(giorno_settimana i)
{
    if (i == giovedì)
        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**?

- No
- Se $i \neq 3$, 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**

- 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 una volta

Ripetuto zero o più volte

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 ;

- Svolgere l'esercizio *stampa_enum.cc* della settimana esercitazione

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

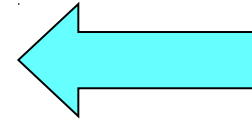
- Svolgere l'esercizio *giorni_lavoro.cc* della settimana esercitazione

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



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

2.4e2 = 2.4*10²

.5

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 $\text{mantissa} * 10^{\text{esponente}}$
 - 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:

$$\text{mantissa}e\text{esponente} = \text{mantissa} * 10^{\text{esponente}}$$

- 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?

- 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
 - Vedremo a breve i valori precisi in gioco

Esercizio

- Svolgere *divis_reale.cc* della settima esercitazione
- Nel caso di numeri con un alto numero di cifre dopo la virgola, o addirittura numeri decimali periodici
 - Viene stampato un alto numero di cifre dopo la virgola?
- Probabilmente no
- Si può controllare tale aspetto della formato di stampa mediante il manipolatore descritto nella prossima slide

Manipolatore setprecision

setprecision(<numero_cifre>)

Setta il massimo numero di cifre per un numero in virgola mobile

- Bisogna includere **<iomanip>**
- L'effettivo output dipende dal formato (generale, scientifico, fisso)
- L'effetto è *persistente*: influenza tutte le prossime operazioni di uscita, fino alla prossima eventuale chiamata di *setprecision*

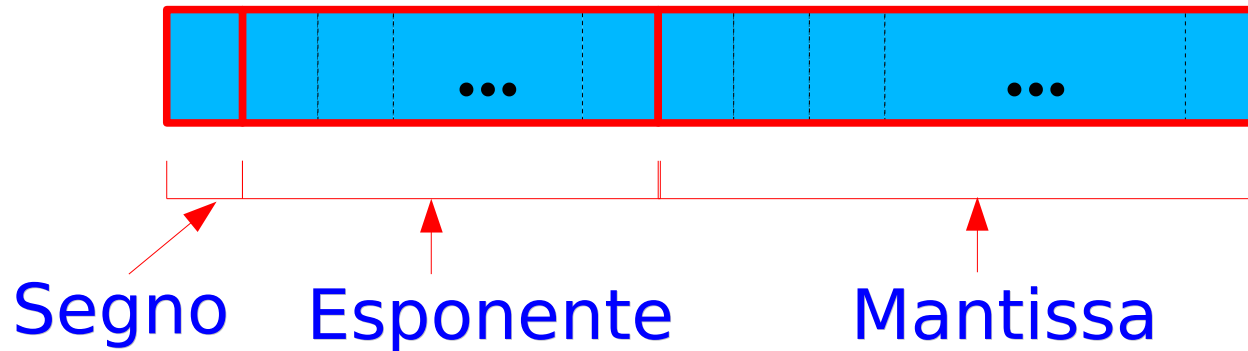
Esercizio

- Utilizzando il manipolatore **setprecision**, modificare la soluzione del precedente esercizio per stampare molte più cifre dopo la virgola

- 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 1/2

- 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, tale che il risultato della somma di sopra è garantito essere maggiore o uguale a 0
 - Questo implica che exp è vincolato ad essere maggiore di $-offset$
 - In particolare, nello standard IEEE 754, si rappresentano esponenti compresi tra $-offset$ e $+(offset-1)$

Offset 2/2

- Quindi, dato il numero *num* memorizzato nel campo esponente
 - L'esponente effettivo è il risultato della seguente sottrazione
 $num - offset$
- Nello standard IEEE 754, il valore massimo per *num* è pari a $+2^{*offset}-1$

Domande

- Quali sono e come vengono memorizzati, con la precedente rappresentazione con *offset*,
 - Esponente minimo rappresentabile
 - Esponente massimo rappresentabile
 - Esponente di valore 0

- Esponente minimo: $-offset$
 - Memorizzato come $-offset + offset = 0$
- Esponente massimo: $+(offset-1)$
 - Memorizzato come $+(offset-1) + offset = 2offset - 1$
- Esponente 0
 - Memorizzato come $0 + offset = offset$

Dettagli sulla mantissa 1/2

- Anche la mantissa differisce leggermente da quella illustrata finora
- Si utilizza una tecnica che permette di risparmiare un ulteriore bit
 - In base 2, la prima cifra della mantissa, dovendo essere diversa da 0, è uguale **esattamente** ad 1
 - Quindi se ne conosce già il valore
 - Allora tale cifra non si memorizza affatto
 - Si memorizzano solo le cifre successive

Dettagli sulla mantissa 2/2

- 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*
- E si memorizzano solo la sequenza di cifre binarie *xxxxxxx* riportata sopra

Esercizio 1/3

- <http://www.h-schmidt.net/FloatConverter/IEEE754.html>
 - Si tratta di numeri a precisione singola (float)
- Esperimenti
 - Provare a settare e resettare il bit del segno
 - Cercare la configurazione di bit che fa sì che l'esponente sia 0
 - Trovare la configurazione di bit che rappresenta il numero 1.5
 - S suggerimenti nella prossima slide

Esercizio 2/3

- Suggestimenti
 - 1.5 è uguale ad $1 + \frac{1}{2}$
 - La prima cifra della mantissa è implicitamente 1
 - In quanto all'esponente, ricordiamoci la banale identità $2^0 = 1$
 - In merito abbiamo appena visto come memorizzare il valore 0 per l'esponente
- Altro esercizio
 - Cercare la rappresentazione di 0.1
 - Suggestimento nella prossima slide

Esercizio 3/3

- Scrivete direttamente 0.1 nel campo “Decimal Representation” e premete invio
- Il valore che viene memorizzato è la rappresentazione esatta di 0.1?
 - Per rispondere alla domanda, guardate il contenuto del campo “After casting to double precision”
 - Torneremo su questo argomento a breve

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 è rappresentabile in modo esatto con tale tipo di dato
 - Indipendentemente da dove si colloca la virgola in tale rappresentazione
- Esempi in base 10
 - un tipo di dato che possa contenere **tutti** i 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 **tutti** i 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	$3.4 \cdot 10^{-38} \dots 3.4 \cdot 10^{38}$
double	15 cifre decimali	$1.7 \cdot 10^{-308} \dots 1.7 \cdot 10^{308}$

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?

- 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 oppure 176471621

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?

- 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 si potrebbe memorizzare in modo esatto
121323231000000000

Domanda

- Come potrebbe essere memorizzato quindi
121323231000000000?

Risposta

- 121323231000000000 è memorizzato come $.121323 \cdot 10^{18}$
- Spiegazione nella prossima slide

Problemi di rappresentazione 1

- Siccome il numero di cifre utilizzate per rappresentare un numero reale è limitato
- Se un numero reale ha più cifre di quelle che si possono rappresentare, allora avviene un **troncamento**
- 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 $\textit{mantissa} * 2^{\textit{esponente}}$ 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 $[0.1]_{10} = [10^{-1}]_{10} = [1/10]_{10} = [1/(2 * \mathbf{5})]_{10} = [???]_2$

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 (perché $10 = 2 * 5$)
- Quindi **non esiste nessuna rappresentazione finita in base 2** di $[0.1]_{10}$
 - Tale numero sarà pertanto **necessariamente memorizzato in modo approssimato**

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
 - Più 1, per la cifra non memorizzata

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

Esercizio

- Svolgere *reale_int.cc* della settima esercitazione

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, in modo esatto, il valore 0 in un float o in un double?

- 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
 - Ossia con una sequenza di bit tutti a 0

- Sulle slide della settimana esercitazione
 - *ascensore.cc*
 - Se non riuscite a realizzare correttamente il programma richiesto in *ascensore.cc*, allora, prima di guardare la soluzione, guardate la prossima slide e riprovate

Confronto approssimato

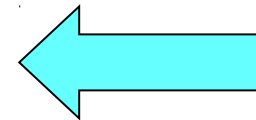
- Ovviamente possono verificarsi errori dovuti al troncamento o all'arrotondamento di alcune cifre decimali anche nell'esecuzione delle operazioni
- In generale, meglio evitare l'uso dell'operatore `==`
 - I test di uguaglianza tra valori reali (in teoria uguali) potrebbero non essere verificati
 - Ad esempio, non sempre vale:
$$(x / y) * y == x$$
- Meglio utilizzare "un margine accettabile di errore":
 - $x == y \rightarrow (x \leq y + \text{epsilon}) \ \&\& \ (x \geq y - \text{epsilon})$
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 (==)

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
 - **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?

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

- 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

Operazioni in modulo unsigned

- Ad esempio, se `MAX_UINT` è una costante di tipo `unsigned int` contenente il valore massimo per il tipo `unsigned int`, allora
 - $\text{MAX_UINT} + 1U = 0$
 - $0 - 1 = \text{MAX_UINT}$
 - $\text{MAX_UINT} + 2U = 1$
 - ...



- 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, ...)

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

numeric_limits<nome_tipo>::epsilon()

minimo valore tale che $1 + \text{epsilon} \neq 1$

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 $m \cdot (2^{\text{esp}})$

numeric_limits<nome_tipo>::min_exponent10

esponente minimo in base 10, cioè valore minimo esp, tale che il numero di possa scrivere nella forma $m \cdot (10^{\text{esp}})$

... 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 $m \cdot (2^{\text{esp}})$

numeric_limits<nome_tipo>::max_exponent10

esponente massimo in base 10, cioè valore massimo esp, tale che il numero di possa scrivere nella forma $m \cdot (10^{\text{esp}})$

- 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 cosa 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

- 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

-
- “... nell' esercizio 4 quando nella consegna dice " il tipo unsigned int è gerarchicamente superiore al tipo int " intende che non può essere viceversa?”

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

```
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 (i==5)
c = d / i;            // evidente perdita di informazione
```

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

- 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

- Per fissare bene i concetti sulle conversioni svolgere, tra gli altri, i seguenti esercizi per casa della settima esercitazione:
 - *divis_reale3.cc*
 - *int_reale_int.cc*
- Finire la settima esercitazione