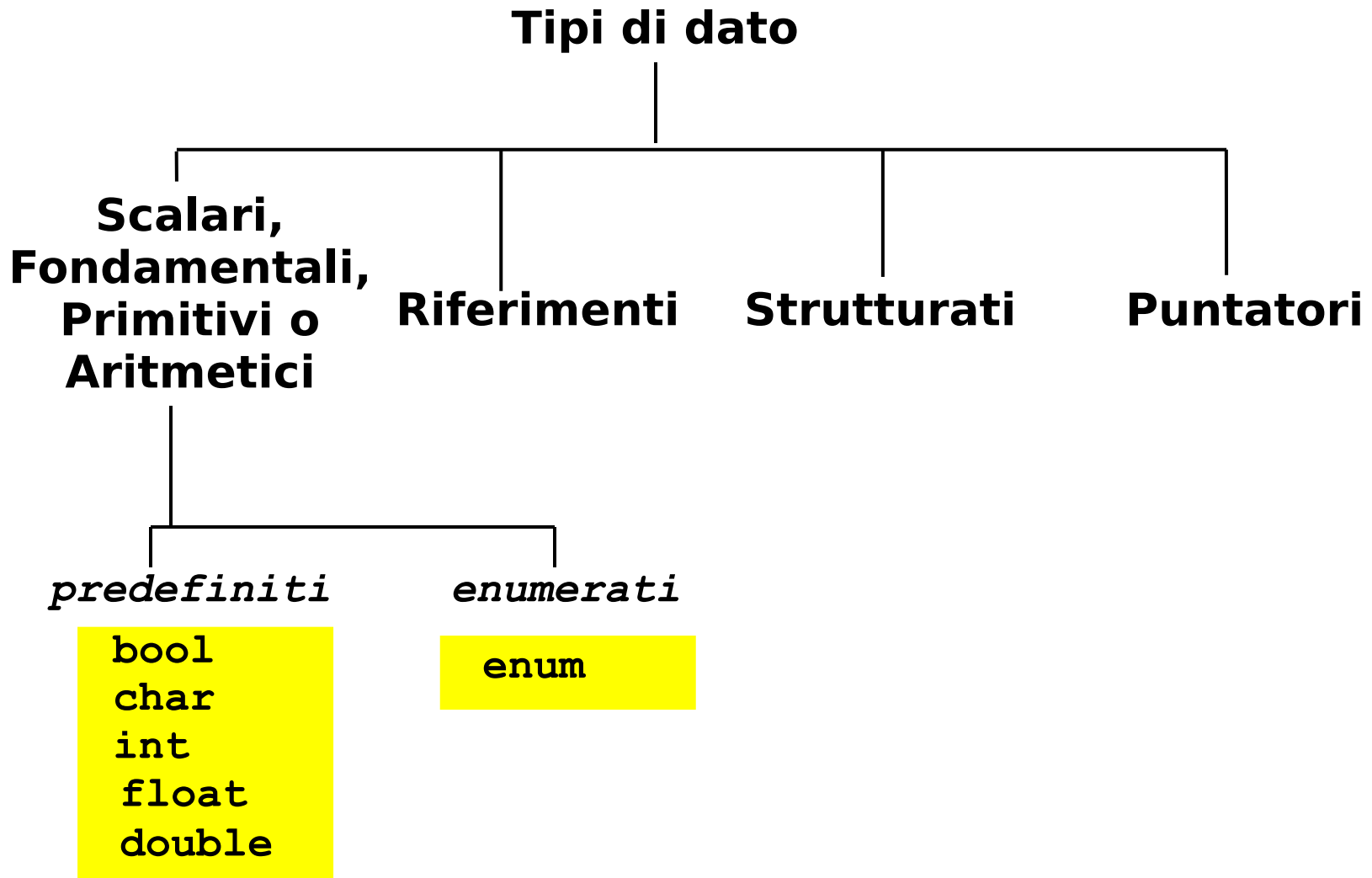


Lezione 7

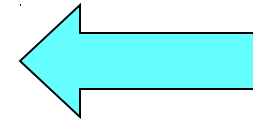
Tipi di dato primitivi
Conversioni di tipo

Tipi di dato



Tipi di dato primitivi

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



Corto-circuito 1/3

- Si dice che un operatore logico binario è **valutato in corto circuito** se
 - il suo secondo operando non è valutato se il valore del primo operando è sufficiente a stabilire il risultato
- In C/C++ sono valutati in **corto-circuito** gli operatori logici **&&** e **||**

Corto-circuito 2/3

- Esempi:

`false && x`

Il valore del primo operando è sufficiente per stabilire che l'espressione è falsa, quindi il secondo operando **non è valutato**

`true || f(x)`

Il valore del primo operando è sufficiente per stabilire che l'espressione è vera, quindi il secondo operando **non è valutato**
Di conseguenza `f(x)` **non è invocata**

`22 || x`

Il valore del primo operando è sufficiente per stabilire che l'espressione è vera, quindi il secondo operando **non è valutato**

Corto-circuito 3/3

- Ricordiamo che `&&` e `||` sono associativi a sinistra, per cui, per esempio:

`a && b && c == (a && b) && c`

`a || b || c == (a || b) || c`

- Ne segue che:

`a && b && c`

Se `a && b` è falso, il secondo operando del secondo `&&` (ossia `c`) non viene valutato

`a || b || c`

Se `a || b` è vero, il secondo operando del secondo `||` (ossia `c`) non viene valutato

- Questo esempio con 3 termini si può banalmente generalizzare al caso di n termini

- Cosa stampa il seguente programma?

```
bool fun() {
    cout<<"fun invocata"<<endl ;
    return true ;
}

main()
{
    bool a = true ;
    if (a && fun())
        cout<<"programma terminato"<<endl ;
}
```

- Stampa:

`fun invocata`

`programma terminato`

- Perché è necessario invocare `fun` per determinare il valore dell'espressione condizionale

- Cosa stampa il seguente programma?

```
bool fun() {
    cout<<"fun invocata"<<endl ;
    return true ;
}

main()
{
    bool a = true ;
    if (a || fun())
        cout<<"programma terminato"<<endl ;
}
```

- Stampa:
`programma terminato`
- Perché **non** è necessario invocare `fun` per determinare il valore dell'espressione condizionale

Espressione condizionale

<condizione> ? <espressione1> : <espressione2>

- Il valore risultante è quello di *<espressione1>* oppure quello di *<espressione2>*
 - Dipende dal valore dell'espressione *<condizione>*:
 - se *<condizione>* è vera, si usa *<espressione1>*
 - se *<condizione>* è falsa, si usa *<espressione2>*
- Esempi:
 - `3 ? 10 : 20` // vale sempre 10
 - `x ? 10 : 20` // vale 10 se x è vero, 20 altrimenti
 - `(x>y) ? x : y` // vale il maggiore fra x ed y

Sintesi priorità degli operatori

Fattori

Termini

Assegnamento

!	++	--	
*	/	%	
	+	-	
>	>=	<	<=
	==	!=	
	&&		
	?:		
	=		

- Svolgere gli esercizi *oper_cond.cc* ed *oper_cond2.cc* della settimana esercitazione

Operatore virgola 1/2

- Date le generiche espressioni $\langle espr1 \rangle$, $\langle espr2 \rangle$, ..., $\langle esprN \rangle$ le si può concatenare mediante l'operatore virgola per ottenere la seguente espressione composta:

$\langle espr1 \rangle, \langle espr2 \rangle, \dots, \langle esprN \rangle$

in cui

- le espressioni $\langle espr1 \rangle$, $\langle espr2 \rangle$, ..., $\langle esprN \rangle$ saranno **valutate l'una dopo l'altra**
- il valore dell'espressione composta sarà uguale a quello dell'ultima espressione valutata

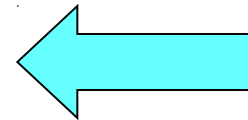
Operatore virgola 2/2

Esempi:

```
int i, j ;  
for (i = 1, j = 3 ; i < 5 ; i++, j--)  
    ... ;
```

Tipi di dato primitivi

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



Prima di iniziare ...

- ... un esempio di quello che si può fare con l'opportuna conoscenza del tipo **char**
- <http://asciimation.co.nz/>
- Non vi preoccupate, cominceremo da qualcosa di più semplice ...

Tipo carattere: char

- Rappresenta l'insieme dei caratteri utilizzabili in accordo allo standard del linguaggio C/C++
- Costanti letterali carattere
 - Dato un carattere, la corrispondente costante letterale carattere si ottiene racchiudendo il carattere tra **singoli apici**

'a' 'b' 'A' '2' '@'

- Diverso dal caso dei letterali numerici, che non andavano corredati da simboli aggiuntivi all'inizio ed alla fine

Caratteri speciali

Mediante le costanti letterali carattere si possono però denotare anche:

- caratteri speciali:

'\n'	A capo	
'\t'	Tabulazione	
'\''	Singolo apice	'
'\\'	Backslash	\
'\"'	Doppi apici	"

Rappresentazione caratteri 1/2

- Abbiamo detto che la memoria è fatta solo di locazioni contenenti numeri
- Come memorizzare un carattere in una locazione che può contenere solo un numero?
- Un problema simile si aveva nelle trasmissioni telegrafiche
 - Si potevano trasmettere solo segnali elettrici
 - Come avevano risolto il problema?

Rappresentazione caratteri 2/3

- Con il codice Morse
 - Associando cioè ad ogni carattere una determinata sequenza di segnali di diversa durata

Rappresentazione caratteri 3/3

- Possibile soluzione per memorizzare caratteri:
 - Associare per convenzione **un numero intero, ossia un codice, diverso a ciascun carattere**
 - Per memorizzare un carattere, si può memorizzare di fatto il numero intero, ossia il codice, che lo rappresenta

Esempio

- Consideriamo solo tre caratteri: a , b e c
- Decidiamo quale numero intero (codice) associare a ciascun carattere, ad esempio

a 1

b 2

c 3

- Per memorizzare, per esempio, il carattere b in una locazione di memoria, vi memorizziamo il numero intero 2
- Se sappiamo che in una data locazione è memorizzato un carattere, controlliamo il numero contenuto nella locazione e dal numero risaliamo al carattere (es.: $2 \rightarrow b$, $3 \rightarrow c$)

Codifica ASCII 1/2

- Generalmente, si utilizza il codice ASCII
- E' una codifica che, nella *forma estesa*, utilizza 1 byte, per cui vi sono 256 valori rappresentabili
 - I codici vanno tipicamente da 0 a 255 (da -128 a +127 nel caso i codici dei caratteri vengano considerati numeri con segno)
- Vi è anche la *forma ristretta* su 7 bit, nel qual caso l'insieme di valori rappresentabili si riduce a 128

Codifica ASCII 2/2

Codice (in base 10)	Carattere
... (0-31, caratteri di controllo)	
32	<spazio>
33	!
34	"
35	#
...	
48	0
49	1
...	
65	A
66	B
67	C
...	
97	a
...	

La tabella completa è reperibile facilmente in rete, e si trova tipicamente anche nell'Appendice dei libri e manuali sui linguaggi di programmazione

Tipo `char`

- Nel linguaggio C/C++ il tipo `char` non denota un nuovo tipo in senso stretto, ma è di fatto l'insieme dei valori interi rappresentabili (tipicamente) su di un *byte*
- Il tipo `char` contiene quindi, di fatto, un sottoinsieme abbastanza piccolo di numeri interi

Intervallo di valori 1/2

<i>Tipo</i>	<i>Dimensione</i>	<i>Intervallo valori</i>
char	1 byte	-127 .. 128 (se considerato con segno) oppure 0 .. 255 (se considerato senza segno)
unsigned char	1 byte	0 .. 255

Intervallo di valori 2/2

- Lo standard non specifica se `char` deve essere considerato con segno o senza
 - La cosa può variare da una macchina all'altra
- Invece `unsigned char` è **sempre senza segno**

Contenuto costanti carattere

- Quindi, le costanti carattere **non denotano altro che numeri interi**
 - Scrivere una costante carattere equivale a scrivere il numero corrispondente al codice ASCII del carattere
 - Ad esempio, scrivere 'a' è equivalente a scrivere 97
 - **Però una costante carattere ha anche associato un tipo, ossia il tipo char**

Stampa di un carattere 1/2

- Di conseguenza, se scriviamo

```
cout<<'a'<<endl ;
```

abbiamo passato il valore 97 al `cout`

- Ma cosa stampa ???
 - Provare per scoprirlo

Stampa di un carattere 2/2

- Stampa un carattere
- Come mai?

- Perché l'operatore `<<` ha dedotto dal tipo (`char`) cosa fare!

- Svolgere *leggi_stamp_char.cc* della settimana esercitazione

Ordinamento 1/2

- I caratteri sono ordinati
- In particolare rispettano il seguente ordinamento (detto lessicografico) per ciascuna delle tre classi (cifre, lettere minuscole, lettere maiuscole):

'0' < '1' < '2' < ... < '9'

'A' < 'B' < 'C' < ... < 'Z'

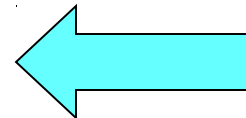
'a' < 'b' < 'c' < ... < 'z'

Ordinamento 2/2

- Ma qual è l'ordinamento **tra** le tre classi
 - Per esempio '1' < 'a'?
- Non è definito!
- Tra le tre classi lo standard del linguaggio non prevede nessuna garanzia di quale dei possibili ordinamenti viene adottato
 - La codifica ASCII ha il suo ordinamento, ma quale codifica deve/può essere utilizzata sulla macchina non è definito dallo standard
 - Lo standard lascia libera la scelta della codifica, purché sia rispettato solo l'ordinamento **all'interno** delle classi
 - L'effettivo ordinamento **tra** le classi dipenderà dalla codifica utilizzata sulla macchina su cui gira il programma

Tipi di dato primitivi

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



Conversioni di tipo

- Dato il valore di una costante, di una variabile, di una funzione o in generale di una espressione
 - Tale valore ha anche associato un tipo
 - Esempio:
 - 2 è di tipo `int`
 - 'a' è di tipo `char`
 - 2<3 è di tipo `bool`
- Esiste un modo per convertire un valore, appartenente ad un certo tipo, nel valore corrispondente in un altro tipo?
 - Sì, uno dei modi è mediante una **conversione esplicita**
 - Esempio: da 97 di tipo `int` a 97 di tipo `char` (ossia la costante carattere 'a')

Conversioni esplicite 1/2

- Tre forme (negli esempi si assuma, ad esempio, che `a` sia una variabile/costante di tipo `char` precedentemente definita):
 - Cast (C/C++)
(*<tipo_di_destinazione>*) *<espressione>*
Esempi: `d = (int) a;`
`fun((int) a) ;`
 - Notazione funzionale (C/C++)
<tipo_di_destinazione>(*<espressione>*)
Esempi: `d = int(a);`
`fun(int(a)) ;`
 - Operatore `static_cast` (solo C++)
`static_cast<<tipo_di_destinazione>>`(*<espressione>*)
Esempi: `d = static_cast<int>(a);`
`fun(static_cast<int>(a)) ;`

Conversioni esplicite 2/2

- In tutti e tre i casi, il valore dell'espressione è convertito nel corrispondente valore di tipo *<tipo_di_destinazione>*, qualche sia il tipo del valore dell'espressione

Operatore `static_cast`

- L'uso dell'operatore `static_cast` comporta una notazione più pesante rispetto agli altri due
- La cosa è voluta
 - Le conversioni di tipo sono spesso pericolose
 - Bisogna utilizzarle solo quando non si riesce a farne a meno senza complicare troppo il programma
 - Un notazione pesante le fa notare di più
- Se si usa lo `static_cast` il compilatore usa regole più rigide
 - Programma più sicuro
- Al contrario con gli altri due metodi si ha piena libertà (di sbagliare senza essere aiutati dal compilatore ...)


```
int i = 100 ;  
char a = static_cast<char>(i) ;
```

- Supponendo che il valore 100 sia rappresentabile mediante il tipo `char`, e che quindi non vi siano problemi di *overflow*
- Che cosa viene memorizzato nell'oggetto di tipo `char`?

- Esattamente il valore 100
- Ma stavolta il valore sarà di tipo `char`
- Similmente, dopo le istruzioni:

```
char a = 100 ; // codice ASCII 100  
int i = static_cast<int>(a) ;
```

nella variabile `i` sarà memorizzato il valore 100, ma il tipo sarà `int`

Domanda

- Supponendo che il codice del carattere 'a' sia 97, che differenza di significato c'è tra le due seguenti inizializzazioni?

```
char b = 'a' ;
```

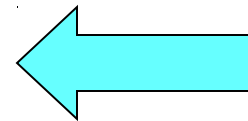
oppure

```
char b = static_cast<char>(97) ;
```

- Nessuna, sono perfettamente equivalenti!

Tipi di dato primitivi

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



- Dalla settimana esercitazione
 - *codice_car.cc*
 - *car_codice.cc*
 - *traccia_car_codici_immediato.cc*
 - Per casa
 - *tabella_ascii.cc*

Operazioni

- Sono applicabili tutti gli operatori visti per il tipo `int`
- Pertanto, si può scrivere:

<code>'x' / 'A'</code>	equivale a	<code>120 / 65</code>	uguale a: <code>1</code>
<code>'R' < 'A'</code>	equivale a	<code>82 < 65</code>	uguale a: <code>false</code> (0 in C)
<code>'x' - '4'</code>	equivale a	<code>120 - 52</code>	uguale a: <code>68</code> (<code>=='D'</code>)
<code>'x' - 4</code>	equivale a	<code>120 - 4</code>	uguale a: <code>116</code> (<code>=='t'</code>)

- Svolgere la settima esercitazione fino alla prova di programmazione inclusa

Caratteri speciali

- Come costante letterale carattere si può specificare direttamente il codice del carattere, con due diverse possibili notazioni:
 - '\nnn' Numero ottale di tre cifre
 - '\0xhhh' Numero esadecimale di tre cifre
- Esempi:
 - '\041' '\0' '\xfa'
- Questa possibilità va usata con molta attenzione, alla luce di quanto spiegato nella prossima slide

Portabilità 1/2

- Ci interessa la forma in cui il numero è memorizzato per poterci lavorare?
 - No, noi lo usiamo semplicemente come un numero intero, pensa a tutto il compilatore
- Soprattutto: se il codice che scriviamo non fa nessuna assunzione su come sono rappresentati i numeri, allora funzionerà su macchine diverse anche se su tali macchine i numeri sono rappresentati in modo diverso
 - Se invece un certo il codice di un programma fa affidamento sul fatto che i numeri sono rappresentati in un certo modo, allora, quando eseguito su una macchina in cui i numeri non sono rappresentati in quel modo, quel programma non funziona più

Portabilità 2/2

- Nel primo caso si dice che il codice è **portabile** tra diverse macchine (architetture), nel secondo caso si dice invece che il codice **non è portabile**
- Lo stesso accade per i codici dei caratteri
 - Dobbiamo scrivere programmi che non facciano assunzioni su quale codifica è utilizzata, altrimenti cambiando codifica i nostri programmi non funzionano più correttamente!
- E' sensato scrivere programmi non portabili solo quando non vi è altra alternativa per il particolare problema da risolvere o per i vincoli temporali imposti
 - In tutti gli altri casi si è fatto semplicemente un cattivo lavoro se si è scritto un programma non portabile

Esercizio per casa

- Esercizio **propedeutico per le prove d'esame**
- Scrivere una funzione che, dato un carattere passato in ingresso (come parametro formale), restituisca il carattere stesso se non è una lettera minuscola, altrimenti restituisca il corrispondente carattere maiuscolo
- Prima di definire il corpo della funzione, scrivere un programma che, usando SOLO tale funzione, legga un carattere da *stdin* e, se minuscolo, lo ristampi in maiuscolo, altrimenti comunichi che il carattere non è minuscolo
 - Adottiamo cioè, per esercizio, un approccio *top-down*

Specifiche della funzione

- Per adottare in modo efficace l'approccio top-down bisogna definire in modo esatto cosa va in ingresso alla funzione e cosa la funzione restituisce
 - Scriviamo quindi solo la dichiarazione della funzione
 - Inseriamo i dettagli sul comportamento della funzione sotto forma di commenti all'intestazione della funzione stessa

Prima parte

```
/*
 * Dato il carattere in ingresso c restituisce il maiuscolo
 * di c utilizzando solo le proprietà di ordinamento dei
 * codici dei caratteri.
 * Assunzione: se c non è minuscolo, ritorna il
 * carattere inalterato.
 */
char maiuscolo(char c);

main() {
    char minus, maius;
    cin>>minus;
    maius = maiuscolo (minus);
    if (minus==maius)
        cout<<"Il carattere "<<minus
            <<" non è minuscolo"<<endl;
    else
        cout<<"Minuscolo = "<<minus<<" - Maiuscolo = "
            <<maius<<endl;
}
```

Bozza di algoritmo funzione

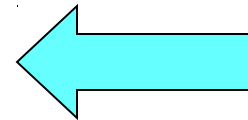
- Se il parametro formale `c` non contiene una lettera minuscola, restituisci il carattere senza alcuna modifica
- Altrimenti, calcola il corrispondente carattere maiuscolo, sfruttando le proprietà di ordinamento della codifica dei caratteri ASCII:
 - ogni carattere è associato ad un valore intero
 - Che noi assumiamo di **NON CONOSCERE**
 - le lettere da 'A' a 'Z' sono in ordine alfabetico
 - le lettere da 'a' a 'z' sono in ordine alfabetico

Funzione

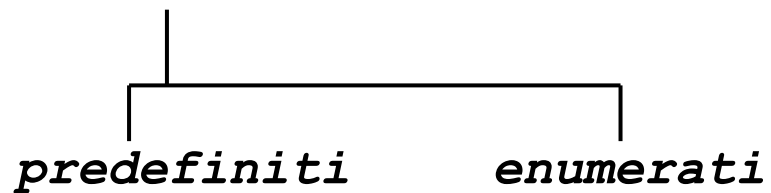
```
/*
 * Dato il carattere in ingresso c restituisce il maiuscolo
 * di c utilizzando solo le proprietà di ordinamento dei
 * codici dei caratteri.
 * Assunzione: se c non è minuscolo, ritorna il
 * carattere inalterato.
 */
char maiuscolo(char c)
{
    if (c < 'a' || c > 'z')
        return c;
    return c - 'a' + 'A';
}
```


Tipi di dato primitivi

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



Tipi di dato



`bool`
`char`
`int`
`float`
`double`

`enum`

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 di un tipo enumerato segue le stesse regole di visibilità di una generica dichiarazione
- Nel campo di visibilità 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 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 1/2

- 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 questo nuovo standard
 - In quanto al gcc, supposto che stiate utilizzando una versione che supporta anche il nuovo standard, avete due possibilità
 - 1) Il compilatore già usa lo standard C++11 di default
 - 2) Il compilatore non usa lo standard C++11 di default
 - Potete verificare se lo standard C++11 è supportato dalla versione del compilatore che state utilizzando, e se è selezionato di default controllando, per esempio, la descrizione dell'opzione `-std` nella pagina di manuale di g++

Utilizzo enum class 2/2

- In particolare, nel gcc lo standard C++11 è denotato come `c++0x`
 - Vecchio nome dello standard C++11
- Se il compilatore supporta lo standard C++11 ma non utilizza di default, dovete aggiungere l'opzione `-std=c++0x` alla riga di comando
- Per evitare problemi di compilazione, nelle soluzioni degli esercizi di questo corso è utilizzato sempre il costrutto `enum` e non `enum class`
- Utilizzate pure il costrutto `enum` e non `enum class` anche nella prova di programmazione

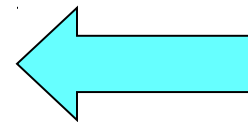
- Svolgere l'esercizio *giorni_lavoro.cc* della settimana esercitazione

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

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



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

- 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`

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

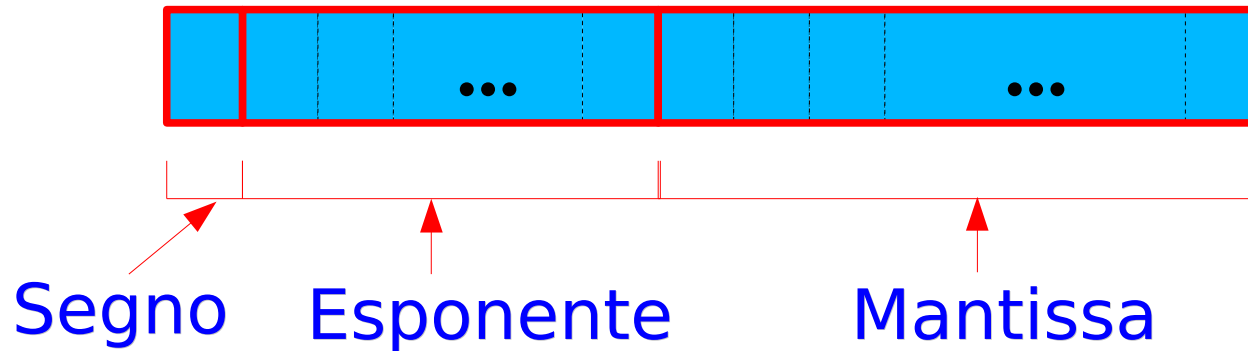
Esercizio

- Svolgere *divis_reale.cc* della settima esercitazione

- 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

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
<code>float</code>	6 cifre decimali	$3.4 \cdot 10^{-38} \dots 3.4 \cdot 10^{38}$
<code>double</code>	15 cifre decimali	$1.7 \cdot 10^{-308} \dots 1.7 \cdot 10^{308}$

Occupazione di memoria:

<code>float</code>	4 byte
<code>double</code>	8 byte
<code>long double</code>	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

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
1213230000000000000 nella forma .121323e18
ma non
1213232310000000000

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

Esercizio

- Svolgere *reale_int.cc* della settima esercitazione

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 $\text{mantissa} * 2^{\text{esponente}}$ 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 l'esercizio *divis_reale2.cc*

- 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

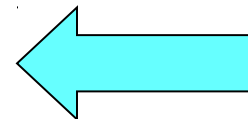
- Ovviamente possono verificarsi errori dovuti al troncamento o all'arrotondamento di alcune cifre decimali anche nell'esecuzione delle operazioni
- In generale, meglio evitare l'uso dell'operatore `==`
 - I test di uguaglianza tra valori reali (in teoria uguali) potrebbero non essere verificati
 - Ad 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

- **Numeri interi** (`int`)
 - Già trattati
- **Valori logici**
 - Già trattati quasi completamente, tranne i seguenti due argomenti, che vedremo in questa lezione:
 - Corto circuito logico
 - Espressione condizionale
- **Caratteri** (`char`)
- **Enumerati** (`enum`)
- **Numeri reali** (`float` e `double`)
- **Tipi e conversioni di tipo**



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>}`

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 quanto ci riguarda 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

- 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

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 1/2

```
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
```

Esempi 2/2

`int i=6, b=5; float f=4.; double d=10.5;`

`d = i;` → assegnamento eterogeneo (double ← int) → 6.
(Converte il valore di i in double e lo assegna a d)

`i=d;` → assegnamento eterogeneo (int ← double) → 10
(Tronca d alla parte intera ed effettua l'assegnamento ad i)

`i=i/b;` → assegnamento omogeneo (int ← int) → 1

`f=b/f;` → assegnamento omogeneo (float ← float) → 1.25
(Converte il b in float prima di dividere, perché f è float)

`i=b/f;` → assegnamento eterogeneo (int ← float) → 1
(L'espressione a destra diventa float perché b è float, tuttavia quando si effettua l'assegnamento, si guarda al tipo della variabile i)

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 decimali 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