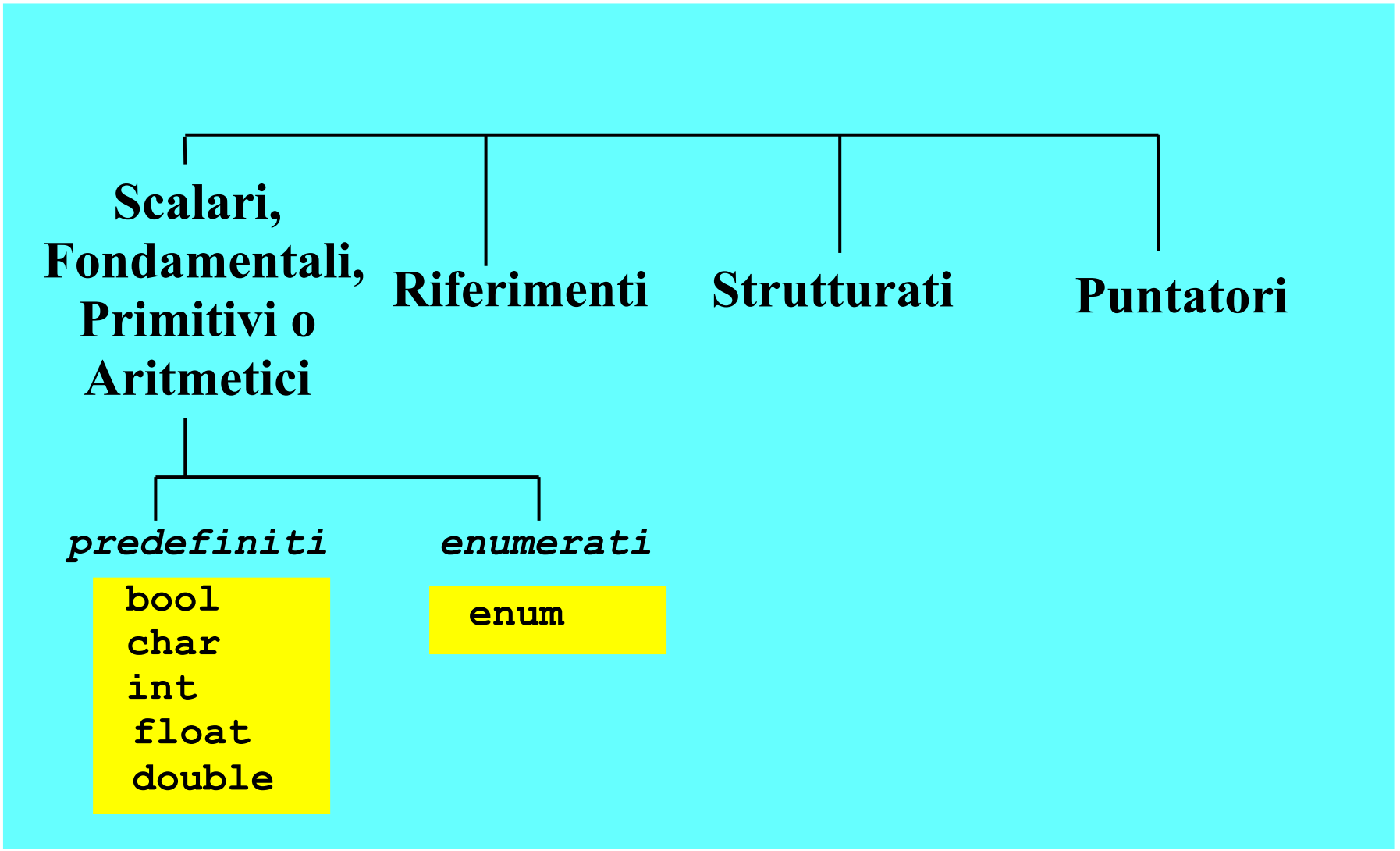
Tipi di dato 'primitivi'

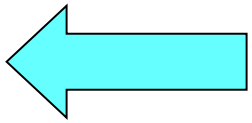
(oltre al tipo int)

Tipi di dato


Scalari, Fondamentali, Primitivi o Aritmetici
Riferimenti
Strutturati
Puntatori
predefiniti
enumerati
bool
char
int
float
double
enum

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **Tipi e conversioni di tipo**



Rivediamo velocemente le cose che già conosciamo sui valori logici in C/C++ ...

Valori logici (*il caso particolare del C*)

In molti linguaggi di programmazione, quali il C++, i valori logici hanno un loro tipo predefinito (come abbiamo visto).

Abbiamo però anche visto che nel linguaggio C, al contrario, non esiste un tipo "logico" come tipo a sé stante!

Pertanto, si sfrutta il tipo "int" (se si vuole si può fare anche in C++):

- il valore 0 (zero) indica **FALSO**
- ogni valore diverso da 0 indica **VERO**

(per convenzione, si tende ad usare 1 per denotare "vero", ma bisogna ricordare che non è così per tutti i compilatori)

Vengono considerati "falsi" anche:

0 '\0' (fine stringa) 0.0 5-5

Vengono invece considerati "veri":

5 'A' 2.35 3*2

Operatori logici in C

operatore logico	*arietà*	C
not (negazione)	*unario*	!
and	*binario*	&&
or	*binario*	||

In C, non esistendo il tipo booleano, gli operatori logici operano su interi e restituiscono un intero:

- il valore 0 viene considerato **falso**
- ogni valore diverso da 0 viene considerato **vero**
- **il risultato è 0 o 1**

Esempi

5 && 7 0 || 33 !5

Tabella di verità degli operatori in C

AND			Ris.	OR			Ris.	NOT	Ris.
1	&&	1	1	1	||	1	1	!1	0
1	&&	0	0	1	||	0	1	!0	1
0	&&	1	0	0	||	1	1		
0	&&	0	0	0	||	0	0		

Torniamo ora a vedere argomenti validi sia per il C che per il C++ ...

Valutazione in *corto-circuito* 1/2

Gli operatori logici *&&* e || sono valutati in **corto-circuito**: la valutazione dell'espressione logica termina non appena si è in grado di determinare il risultato

→ I termini successivi sono valutati ***solo se necessario***

Esempi

22 || x	L'espressione è già vera in partenza (22 è vero, stile C), il secondo termine **non è valutato**
0 && x	L'espressione è già falsa in partenza (0 è falso, stile C), il secondo termine **non è valutato**
true || f(x)	L'espressione è già vera in partenza, il secondo termine non è valutato, quindi f(x) **non è invocata**

Valutazione in *corto-circuito* 2/2

Espressioni logiche composte:

a && b && c	Se a && b è falso, il secondo && **non viene valutato**
a || b || c	Se a || b è vero, il secondo || **non viene valutato**

Operatori di manipolazione bit a bit

Operano solo su tipi "int" e "char"

- **&** AND
- **|** OR
- **^** XOR (OR esclusivo)
- **~** complemento
- **<<** SHIFT A SINISTRA *Moltiplicazione per potenza di 2*
- **>>** SHIFT A DESTRA *Divisione per potenza di 2*

Esempi

a = 4**<<**3 → **4*2³** → 32

b = 10**>>**2 → **10/2²** → 2

Non si utilizzeranno in questo corso

Espressioni condizionali
(operatore condizionale **?**)

condizione ? espr1 : espr2

Il valore risultante è o quello di *espr1*, o quello di *espr2*: dipende dal valore dell'espressione condizione:

- se condizione denota vero (o un valore ≠ zero), si usa espr1
- se condizione denota falso (o il valore zero), si usa espr2

Esempi

3 ? 10 : 20	denota sempre 10
x ? 10 : 20	denota 10 se x è *true* o diversa da 0, altrimenti 20
(x>y) ? x : y	denota il maggiore fra x e y

Sintesi priorità degli operatori

Fattori

Termini

Assegnamento

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

Esercizi

- *oper_cond.cc*
- *oper_cond2.cc*

Operatore virgola

espr1, espr2, ..., esprN

Date le generiche espressioni espr1, espr2, ..., esprN le si può concatenare mediante l'operatore virgola. Si ottiene un'espressione composta in cui

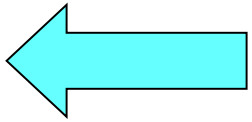
- le espressioni espr1, espr2, ..., esprN saranno valutate l'una dopo l'altra
- il valore di ritorno dell'espressione composta sarà uguale a quello dell'ultima espressione valutata

Esempio:

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

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **Tipi e conversioni di tipo**



Tipo Carattere: char

- Rappresenta l'insieme dei caratteri disponibili nel sistema
- Costanti (letterali) carattere:

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

- L'insieme dei caratteri è **ordinato**
- L'ordine dipende dalla codifica adottata ...

Rappresentazione dei caratteri

- Abbiamo detto che la memoria è fatta solo di locazioni contenenti numeri
  - Come memorizzare un carattere?
- Un problema simile si aveva nelle trasmissioni telegrafiche
  - Si potevano trasmettere solo segnali elettrici
  - Idea: codice Morse, ad ogni carattere è associata una determinata sequenza di segnali di diversa durata
- Possibile soluzione per memorizzare caratteri:
  - Associare per convenzione un numero diverso a ciascun carattere
  - Per memorizzare un carattere, si può memorizzare di fatto il numero che lo rappresenta

Codifica ASCII

- Generalmente, si utilizza il **codice ASCII**
- E' una codifica che (nella *forma estesa*) utilizza 1 byte, per cui vi sono 256 valori rappresentabili

- Vi è anche la *forma ristretta* su 7 bit, nel qual caso l'insieme di valori rappresentabili si riduce a 128

Codifica ASCII

Codice (decimale)	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 si trova tipicamente nell'Appendice dei libri e manuali sui linguaggi di programmazione

Tipo *char*

- Nel linguaggio C/C++ (a differenza di altri linguaggi) 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
- Quindi, le costanti carattere non denotano altro che numeri
  - Scrivere una costante carattere equivale a scrivere **il numero corrispondente** al codice ASCII del carattere
    - Ad esempio, scrivere `'a'` è numericamente 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 ???

Stampa di un carattere 2/2

- Si può verificare che stampa un carattere, come mai?
  - Perché ha dedotto cosa fare dal tipo

Ordinamento

- Nelle operazioni, bisogna inoltre considerare che le tabelle ASCII dei caratteri occidentali rispettano il seguente ordinamento (detto *lessicografico*):

? ? ? ?

‘0’<‘1’<‘2’<…<‘9’<…<‘A’<‘B’<‘C’<…<‘Z’<…<‘a’<‘b’<‘c’<…<‘z’

- Vale la regola di **prossimità** all’interno di ciascuna di queste tre classi ovvero, il carattere successivo di ‘a’ è ‘b’, di ‘B’ è ‘C’, di ‘4’ è ‘5’, ecc.
- **Tuttavia, le tre classi (minuscole, maiuscole, numeri), per quanto disgiunte, non sono contigue e nessuna garanzia tra l'ordinamento tre le classi!**

Tipo char (*cont.*)

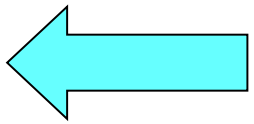
Tipo	*Dimensione*	*Valori*
char	1 byte	-127 .. 128
unsigned char	1 byte	0 .. 255

Sono quindi applicabili tutti gli operatori visti per gli “int”.
Pertanto, si può scrivere:

‘x’ / ‘A’	equivale a	120 / 65	uguale a: 1
‘R’ < ‘A’	equivale a	82 < 65	uguale a: false (0 in C)
‘x’ – ‘4’	equivale a	120 – 52	uguale a: 68 (==‘D’)
‘x’ – 4	equivale a	120 – 4	uguale a: 116 (==‘t’)

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **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
  - Es: *'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
  - Es: da 97 di tipo *int* a 97 di tipo *char* (ossia la costante carattere *'a'*)

Conversioni esplicite

- Tre modalità:

  – ***Cast*** ***(C/C++)***

  (<tipo>) espressione → Es: d=(int) a;
  fun((int) b) ;

  – **Notazione funzionale** ***(C/C++)***

  <tipo>(espressione) → Es: d=int(a);
  fun(int(b)) ;

  – **Operatore *static_cast*** ***(solo C++)***

  static_cast<tipo>(espressione)
  → Es: d=static_cast<int>(a);
  fun(static_cast<int>(b)) ;

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
  - In generale è meglio evitarle
  - 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

Esempio

int i = 100 ;

char a = static_cast<char>(i) ;

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

Risposta

- 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

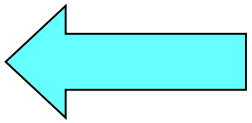
```
char b = static_cast<char>(97) ;
```

Risposta

- Nessuna, sono perfettamente equivalenti

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **Tipi e conversioni di tipo**



Esercizi

- **Dai lucidi di esercitazione:**
  - ***car_codice.cc***
  - ***codice_car.cc***
  - ***tabella_ascii.cc***

Esercizio (*Specifica*)

- **Scrivere una funzione che, dato un carattere, 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**
  - **Procedere in questo modo è un esempio di approccio top-down**

Esercizio (*Idea*)

char **maiuscolo**(char 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>

```
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;
 }
```

Esercizio (*Algoritmo*)

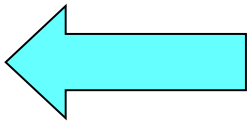
- Se **c** non è una lettera minuscola, restituisci il carattere senza alcuna modifica.
- Altrimenti, calcola il corrispondente carattere maiuscolo, sfruttando le proprietà di ordinamento della codifica dei caratteri ASCII:
  – ogni carattere è associato ad un intero
  – le lettere da ‘A’ a ‘Z’ sono in sequenza
  – le lettere da ‘a’ a ‘z’ sono in sequenza

Esercizio (*Programma*)

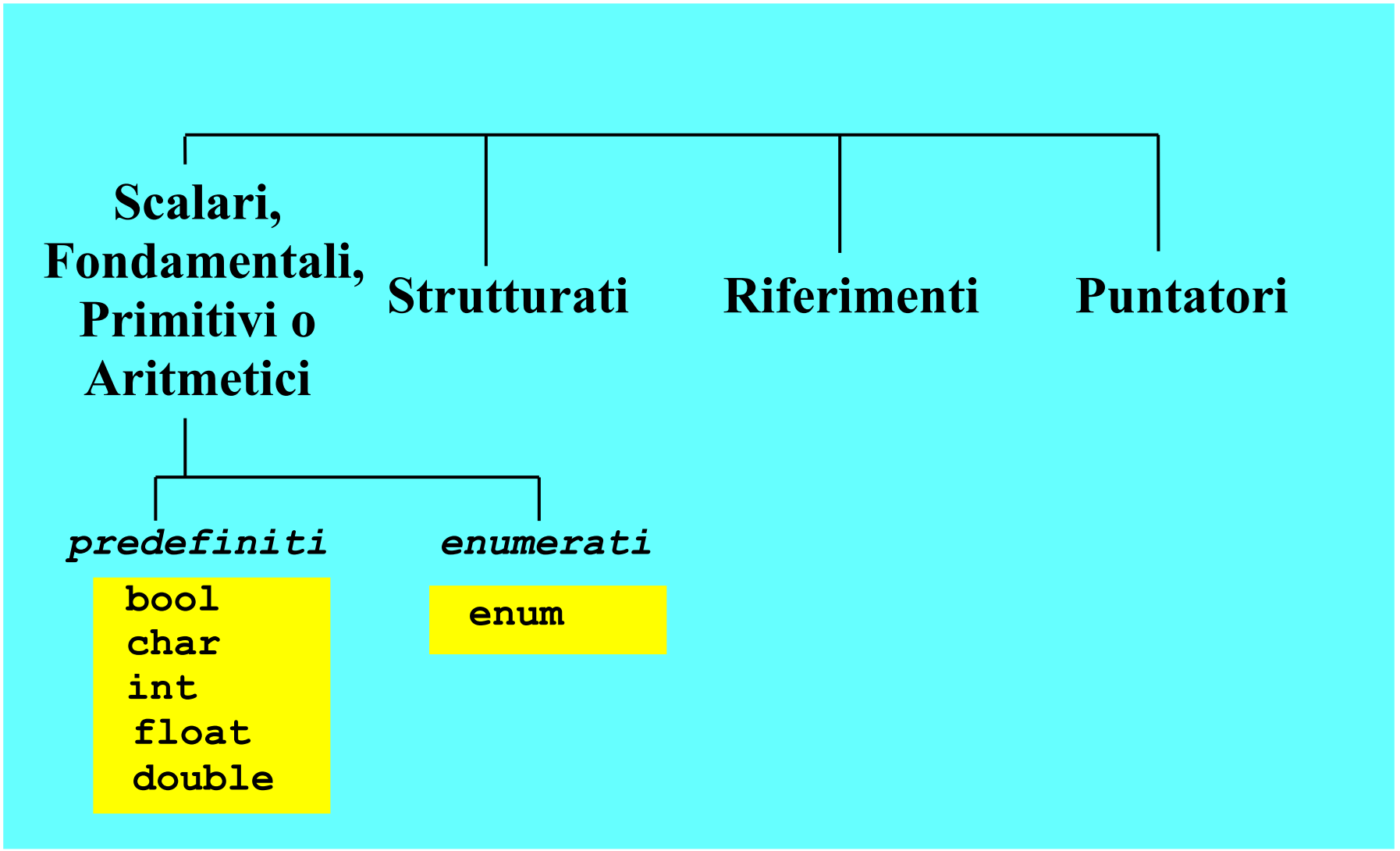
```
#include <iostream>
char maiuscolo (char);                    /* Prototipo di funzione */

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

/*  Funzione che dato un carattere in ingresso,ritorna il corrispondente
    carattere maiuscolo se si tratta di un carattere minuscolo */
char maiuscolo(char c)                    /* Definizione di funzione*/
{ if (c<'a' || c>'z')   return c;
  return c – 'a' + 'A';
 }
```

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **Tipi e conversioni di tipo**



Tipi di dato


Scalari, Fondamentali, Primitivi o Aritmetici
Strutturati
Riferimenti
Puntatori
predefiniti
enumerati
bool
char
int
float
double
enum

Tipo enumerato 1/2

- **Insieme di costanti** definito dal programmatore
  - ciascuna individuata da un identificatore e detta **enumeratore**
- Esempio di dichiarazione:
  enum colori_t {rosso, verde, giallo} ;
  - dichiara un tipo di nome colori_t e tre costanti (enumeratori) di nome *rosso*, *verde* e *giallo*
  - vedremo a breve quali valori sono assegnati automaticamente alle costanti se non specificato dal programmatore

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
    - Esempio: così come si può scrivere
      int a ;
      si potrà anche scrivere
      colori_t a ;
      il cui significato è quello di definire un oggetto di nome *a* e di tipo *colori_t*
  - i valori possibili di oggetti di tipo *colori_t* saranno quelli delle costanti *rosso*, *verde* e *giallo*
    - Quindi l'oggetto *a* definito sopra potrà assumere solo i valori *rosso*, *verde* e *giallo*

Sintassi

- Dichiarazione di un tipo enumerato

<dichiarazione_tipo_enumerato> ::=
enum <identificatore> {<lista_dich_enumeratori>} ;
<lista_dich_enumeratori> ::=
<dich_enumeratore> **{**, <dich_enumeratore>**}**
<dich_enumeratore> ::=
identificatore **[**= <espressione>**]**

Ripetuto 0 o più volte

Ripetuto 0 o una volta

Inizializzazione e visibilità

- Agli enumeratori sono associati per default valori interi consecutivi a partire da 0
  Es: 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
    Es: *colori_t c ;*
    colori_t d = rosso ;

Esercizio

- Dalle slide di esercitazione:
  - *stampa_enum.cc*

Occupazione di memoria e range

- **Stessa occupazione di memoria (in numero di byte) e stessi operatori del tipo *int***

  Intervallo (*logico*!) limitato all'elenco dei valori.
- Ma non c'è controllo sull'intervallo da parte del compilatore
  - Finché si usano solo gli enumeratori non ci sono problemi
  - Inoltre:
    int a = 100; colore_t c = a ;
    genera correttamente un errore a tempo di compilazione
  - ma sono possibili cose pericolose tipo:
    int a = 100; colore_t c = static_cast<colore_t>(a) ;

Note sui tipi enumerati (1)

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.

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

Un tipo enumerato è **totalmente ordinato.** Su un dato di tipo enumerato sono applicabili tutti gli **operatori relazionali**:

lu < ma	→	vero
lu >= sa	→	falso
rosso < giallo	→	vero

Note sui tipi enumerati (2)

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

  enum Mesi {**gen=1**, feb, mar, ... } ;

  gen → 1, feb → 2, ecc.

  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 ;
  Es.: enum {rosso, verde, giallo} colore ;
- Nel campo di visibilità della variabile è possibile utilizzare sia la variabile che gli enumeratori dichiarati nella sua definizione

Esercizio

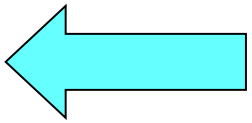
- Dalle slide di esercitazione
  - *giorni_lavoro.cc*

Benefici del tipo Enumerato

1) Decisamente migliore leggibilità
2) Indipendenza dai valori esatti e dal numero di costanti
   - Conseguenze importantissime:
     - se cambio il valore di una costante, non devo modificare il resto del programma
     - posso aggiungere nuove costanti senza dover necessariamente modificare il resto del programma
3) Maggiore robustezza agli errori
   - Se si usano solo gli enumeratori non è praticamente possibile usare valori sbagliati

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **Tipi e conversioni di tipo**



Numeri Reali

Esistono due modi per rappresentare un numero reale:

1. **Virgola fissa** — **Numero di cifre intere e decimali deciso a priori**

2. **Virgola mobile** — **Numero di cifre intere e decimali non deciso a priori**
   - Tipicamente però si decide a priori il numero **totale** massimo di cifre
   - Ad esempio, se il numero totale massimo di cifre fosse 5, potremmo scrivere i numeri
     3.14 12.345 475.22 .32412
     ma non potremmo scrivere
     123.456 1.321445 .987276
     perché eccederemmo il numero totale di cifre

Numeri in virgola mobile 1/2

- Si decide a priori il numero massimo di cifre perché questo permette una rappresentazione abbastanza semplice dei numeri reali
- Tipicamente un numero reale è rappresentato mediante tre componenti:
  - Segno
  - Mantissa (*significand*), ossia le cifre del numero
  - Esponente in base 10, ossia il numero si immagina nella forma
    $mantissa * 10^{esponente}$
  - Tipicamente la mantissa è intesa come un numero a virgola fissa, con la virgola subito prima (o in altre rappresentazioni subito dopo) la prima cifra

Numeri in virgola mobile 2/2

- Una notazione che torna utile per evidenziare le precedenti componeti di un numero reale è la *notazione scientifica*:

  *mantissa*e*esponente*
- Esempi:

Numero	Notazione Scientifica	Segno	Mantissa	Esponente
123	.123e3	+	.123	3
12.3	.123e2	+	.123	2
0.123	.123e0	+	.123	0
-1.23	-.123e1	-	.123	1

Tipi *float* e *double*

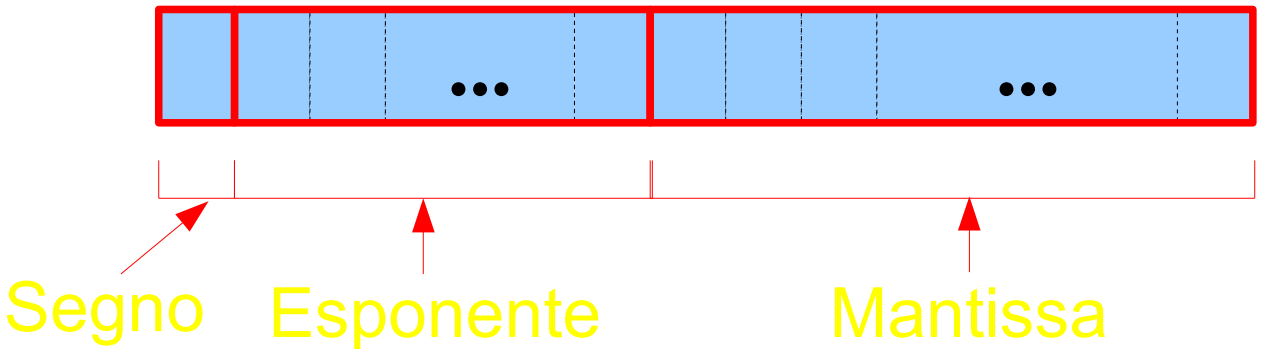
- **Nel linguaggio C/C++ i numeri reali sono rappresentati mediante i tipi *float* e *double***
  - **Sono numeri in virgola mobile**
  - **Mirano a rappresentare (*con diversa precisione*) un sottoinsieme dei numeri REALI**
  - I tipi *float* e *double* (così come *int* per gli INTERI), sono solo un'approssimazione dei numeri reali, sia come **precisione** (ossia valore assoluto minimo rappresentabile) sia come **intervallo** di valori rappresentabili

IEEE 754

- I numeri *float* e *double* sono tipicamente rappresentati/ memorizzati in base 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 nella forma
  $mantissa * 2^{esponente}$

Rappresentazione in memoria

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



- Tale sequenza di bit è tipicamente distribuita su più celle contigue in memoria

Tipi *float* e *double* (*cont.*)

STANDARD COMUNE

(ma non necessariamente vero per tutte le architetture)

float	4 byte
double	8 byte
long double	10 byte

Tipo	Precisione	Valori
float	6 cifre decimali	3.4^10-38 ... 3.4 ^ 10+38
double	15 cifre decimali	1.7 ^ 10-308 ... 1.7 ^ 10+308

Problemi di rappresentazione 1/2

- Siccome il numero di cifre utilizzate per rappresentare un numero reale è limitato, si potrebbero verificare approssimazioni (*troncamenti* o *arrotondamenti*) nella rappresentazione di un numeri reale con molte cifre

Esempio

Il numero 277290.0010044

se si avessero massimo 10 cifre a disposizione potrebbe essere rappresentato come 0.277290001e+6

Tuttavia, questa rappresentazione trasformerebbe il numero originario

277290.0010044 → 277290.001

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 2/2

- Il numero di cifre limitato non è l'unica fonte di problemi di rappresentazione
- Ad esempio, come si può rappresentare 0.1 nella forma *mantissa* * $2^{esponente}$ con la mantissa rappresentata in base 2?
- Sarebbe necessario poter scrivere il numero nella forma $m*2^0$ ove m dovrebbe essere un numero minore di 1 rappresentato in base 2
  - 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 $[0.1]_2 = 1 * 2^{-1}$, $[0.01]_2 = 1 * 2^{-2}$
    - Ma $[0.1]_{10} = [???]_2$

Risposta

- Purtroppo solo i numeri razionali che hanno una potenza di 2 al denominatore si possono esprimere con una sequenza finita di cifre binarie
- Quindi non esiste nessuna rappresentazione in base 2 di $[0.1]_{10}$
  - Tale numero sarà pertanto necessariamente memorizzato in modo approssimato

Numeri reali in un programma C/C++

Si possono utilizzare i seguenti formati:

24.0 .5 2.4e1 240.0e-1

- La notazione scientifica può tornare molto molto utile per scrivere numeri molto grandi o molto piccoli
- Per indicare che un numero è da intendersi come reale anche se non ha cifre dopo la virgola, si può terminare il numero con un punto
  Es.: **123.**

Operatori su *float* e *double*

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

Divisione tra reali ed interi

- Se si prova ad eseguire la divisione tra un oggetto di tipo *int* o *char* ed un oggetto di tipo reale, si effettua di fatto la divisione reale
- Vedremo in seguito il motivo ...

Esercizi

- Sulle slide di esercitazione
  - Da *divis_reale.cc* ad *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

Attenzione!

- A causa della rappresentazione su di un numero finito di cifre, ci possono essere errori dovuti al troncamento o all'arrotondamento di alcune cifre decimali anche nelle operazioni

- Meglio evitare l'uso dell'operatore **==**

  in quanto i test di uguaglianza tra valori reali (in teoria uguali) potrebbero non essere verificati.

  Ovvero, non sempre vale: **(x / y) *y == x**

- Meglio utilizzare "un margine accettabile di errore":

  (x == y) $\rightarrow$ (x <= y+epsilon) && (x >= y-epsilon)

  dove, ad esempio, **const float** epsilon=0.000001

Conversione da reale ad intero

- La conversione da reale ad intero è tipicamente effettuata per troncamento
- Si conserva cioè solo la parte intera del numero di partenza
  - Ovviamente a meno di problemi di overflow

Esercizi ...

Dove si sbaglia frequentemente …

Operazioni matematiche e tipi di dato

- Divisione fra interi e divisione fra reali
  (stesso simbolo **/**, ma differente significato)
- Significato e uso dell'operazione di modulo (**%**), che non è definita per i numeri reali
- Operatore di assegnamento (**=**) e operatore di uguaglianza (**==**)
- Notazione prefissa e postfissa di **++** e **- -** negli assegnamenti

Riassunto

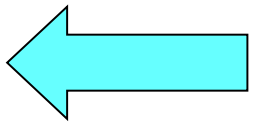
- Valori logici
- Valutazione in corto-circuito

- Tipo char (in pratica “simile” a int)
- Codifica ASCII

- Tipo float e double

Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Enumerati**
- **Reali**
- **Tipi e conversioni di tipo**



Elenco tipi di dato in C/C++

- **Tipo intero**
  - **int** **(32 bit)**
  - **short int (o solo short)** **(16 bit)**
  - **long int (o solo long)** **(64 bit)**
- **Tipo naturale**
  - **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

Elenco tipi di dato in C/C++

- **Tipo carattere**
  - **char** **(8 bit)**
  - **signed char** **(8 bit)**
  - **unsigned char** **(8 bit)**
  - **A seconda delle implementazioni char è implicitamente signed (può avere anche valori negativi) o unsigned**

- **Tipo reale**
  - **float**
  - **double**
  - **long double**

Elenco tipi di dato in C/C++

- **Tipo booleano**
  - **bool**

- **Tipo enumerazione**
  - **enum nome_tipo {<lista_nomi_costanti>}**

Limiti 1/2

In C++, includendo <**limits**> si ha accesso alle seguenti informazioni

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*, ...)

Limiti 2/2

(le seguenti informazioni hanno significato per i numeri in virgola mobile):

numeric_limits<nome_tipo>::epsilon()

valore positivo minimo *epsilon* tale che 1 + *epsilon* != 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*(2^*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*(10^*esp*)

Limiti 3/2

... 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*(2^*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*(10^*esp*)

- Esercizio: *limiti.cc*

Tipizzazione dei dati del C/C++

- Non ci sono problemi di interpretazione fin quando in una espressione tutti i fattori sono dello stesso tipo
- **Ma cosa succede se un'espressione con risultato di tipo *int* viene assegnata ad una variabile di tipo *float* o viceversa?**
- **E se un operatore binario viene invocato con due argomenti di tipo diverso?**

Conversioni di tipo

- Nei precedenti casi si hanno due possibilità:
  1. Si inseriscono **conversioni esplicite** per rendere le espressioni omogenee
  2. Non si inseriscono 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

Coercion

- **Il C/C++ è un linguaggio a "tipizzazione forte"**
  - **Ossia il compilatore controlla ogni operazione per evitare inconsistenze nel tipo di dato o perdite di informazione**
- In generale, le conversioni implicite di tipo che non provocano perdita di informazione sono automatiche
- Tuttavia, le conversioni implicite che possono provocare perdita di informazioni non sono illegali
  - vengono tipicamente segnalate da ***warning***

- In generale le conversioni implicite avvengono a tempo di compilazione in funzione di un ben preciso insieme di regole

Coercion (*cont.*)

Regole per **operandi eterogenei**

1. **Se un operatore binario ha operandi eterogenei,**
   - Ogni operando *char* o *short* viene convertito **comunque** in *int*
   - Se dopo l'esecuzione del passo precedente, l'espressione è ancora eterogenea rispetto agli operandi coinvolti, si converte temporaneamente l'operando di tipo inferiore facendolo diventare di tipo superiore. La gerarchia è:

CHAR < INT < UNSIGNED INT < LONG INT < UNSIGNED LONG INT < FLOAT < DOUBLE < LONG DOUBLE

O più sinteticamente

CHAR < INT < FLOAT < DOUBLE < LONG DOUBLE

- A questo punto l'espressione risulta **omogenea** e viene invocata l'operazione opportuna. Il risultato sarà dello stesso tipo.
- L'effettiva operazione effettuata sarà quindi quella relativa all'operando con più alto livello gerarchico.

Esempi Coercion - espressioni

int a, b, c; float x, y; double d;

a*b+c → espressione omogenea (**int**)

a*x+c → espressione eterogenea (**float**): **a** *è convertito in float*

x*y+x → espressione omogenea (**float**)

x*y+5-d → espressione eterogenea (**double**): ***x*y+5*** *passa tutto in float e poi 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***

Coercion (*cont.*)

Regole per **assegnamenti eterogenei**

- **L'espressione a destra dell'assegnamento viene valutata come descritto dalle regole per la valutazione del tipo di un'espressione omogenea o eterogenea**

- **Per determinare il tipo del valore assegnato si deve considerare poi il tipo della variabile a sinistra dell'assegnamento:**
  - Se il tipo della variabile è **gerarchicamente uguale o superiore** al tipo dell'espressione da assegnare, l'espressione viene convertita nel tipo della variabile probabilmente senza perdita di informazione.
  - Se il tipo della variabile è **gerarchicamente inferiore** al tipo dell'espressione da assegnare, l'espressione viene convertita nel tipo della variabile con i conseguenti rischi di perdita di informazione (dovuti ad un numero inferiore di byte utilizzati oppure ad una diversa modalità di rappresentazione).

Esempi Coercion - assegnamenti

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 → int → double      d==75. */

i = d;      /* sicuro troncamento di informazione della
               parte decimale (i==5), ma in generale anche
               rischio di perdita di informazione della parte
               intera */

c = d / i;  /* (elevato rischio di) perdita di informazione  */
```

Esempi Coercion – assegnamenti (*cont.*)

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 ***x*** è float, tuttavia quando si effettua l'assegnamento, si guarda al tipo della variabile ***i***)

Conversione esplicita - *Esercizio*

int a, b=2; **float x=5.8, y=3.2;**

```
a = static_cast<int>(x) % static_cast<int>(y);
a = static_cast<int>(sqrt(49));
```

a = b + x; è equivalente a:

y = b + x; è equivalente a:

a = b + int(x+y); è equivalente a:

a = b + int(x) + int(y); è equivalente a:

Conversione esplicita - *Soluzioni*

```
int a, b=2;   float x=5.8, y=3.2;

a = static_cast<int>(x) % static_cast<int>(y);     →  2
a = static_cast<int>(sqrt(49));                    →  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 + int(x+y); è equivalente a:

```
a=b+static_cast<int>(9.0);                      →  11
```

a = b + int(x) + int(y); è equivalente a:

```
a=b+static_cast<int>(5.8)+static_cast<int>(3.2);
                          →  a= 2+8 →  10
```

Conversione esplicita
Esempi di perdita di informazione

```
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);
                          /* 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 in fase di compilazione!