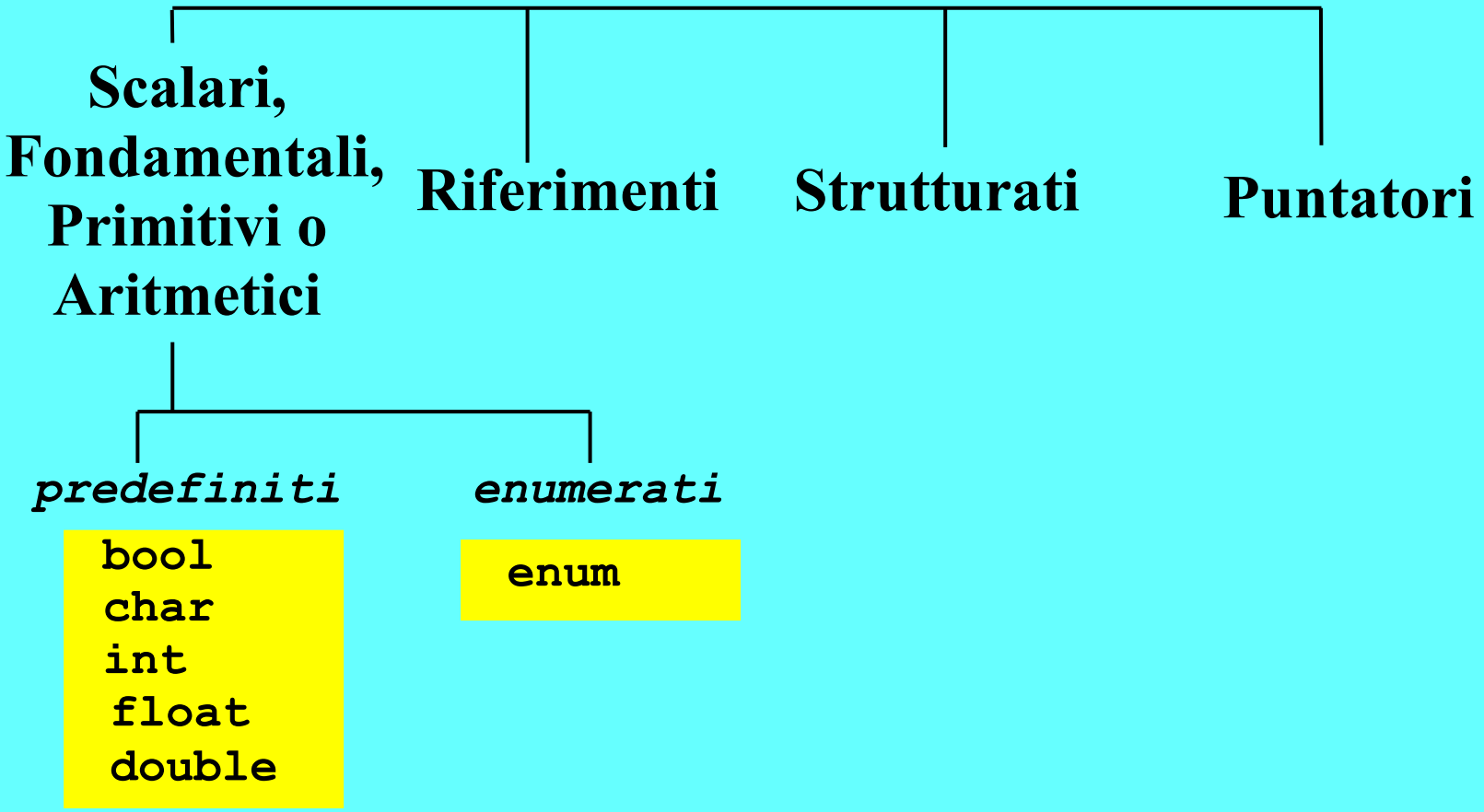
# Tipi di dato “primitivi”

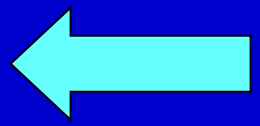
## (oltre int)

# 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**
- **Reali**
- **Enumerati**
- **Tipi e conversioni di tipo**



# Valori logici (*il caso particolare del C*)

In molti altri linguaggi di programmazione, quali il C++, i valori logici hanno un loro tipo predefinito.

**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 EOF (end of file)

Vengono invece considerati "veri":

5 'A' 2.35 3*2

# Operatori logici

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

In C, non esistendo il tipo boolean, 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

| 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 | | |

# Valutazione in *corto-circuito* 1/2

**Gli operatori logici && e || sono valutati in corto-circuito: la valutazione dell'espressione logica termina 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 → **$4*2^3$** → 32

b = 10**>>**2 → **$10/2^2$** → 2

# 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

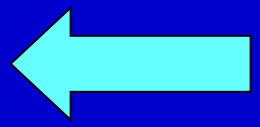
| |
|---|
| ! ++ -- |
| * / % |
| + - |
| > >= < <= |
| == != |
| && |
| \|\| |
| ? : |
| = |

# Esercizi

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

# Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Reali**
- **Enumerati**
- **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

- 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, memorizziamo 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, per cui l'insieme di valori rappresentabili si riduce a 128

# Codifica ASCII

| Codice (decimale) | Carattere |
|---|---|
| ... (0-31, caratteri di controllo) | |
| 32 | <bianco> |
| 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 equivalente al dominio dei **valori interi** rappresentati su di un byte
- Quindi, ad esempio, scrivere 'a' da qualche parte nel programma, equivale a scrivere il numero corrispondente al codice ASCII della lettera *a*
  - In particolare scrivere 'a'  è numericamente equivalente a scrivere 97
  - Però 'a' ha anche associato un **tipo**, ossia il tipo *char*
- Di conseguenza, se scriviamo
  *cout<<'a'<<endl ;*
  abbiamo passato il valore 97 al cout
- Ma cosa stampa ???

# Tipo “char”

- Come mai ha stampato un carattere?
  - Perché ha dedotto cosa fare dal tipo
- 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) non sono contigue!**

# Tipo “char” (*cont.*)

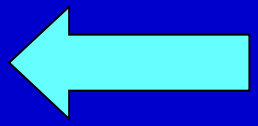
| *Tipo* | *Dimensione* | *Valori* |
|---|---|---|
| **char** | 1 byte | -127 .. 128 |
| **unsigned char** | 1 byte | 0 .. 255 |

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: 0 |
| ‘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**
- **Reali**
- **Enumerati**
- **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)**

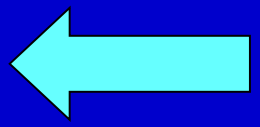
  Es: d=static_cast<int>(a);
  fun(static_cast<int>(b)) ;

# 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

# Tipi di dato primitivi

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



# **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**

# **Esercizio** (*Idea*)

char **maiuscolo**(char c);

< restituisce il maiuscolo di c utilizzando la tabella ASCII

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*)

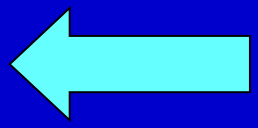
```
#include <iostream>
char maiuscolo (char);                    /* Prototipo di funzione */

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 minuscolo lo trasforma in maiuscolo */
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**
- **Reali**
- **Enumerati**
- **Tipi e conversioni di tipo**



# Numeri Reali

Esistono due modi per rappresentare un numero reale:

1. **Virgola fissa** **Numero di cifre intere e decimale deciso a priori**
   P.es., 0.0015 3.14 123.27 52. .23

2. **Virgola mobile** **Numero totale di cifre deciso a priori**
   - Mantissa (tipicamente con il primo decimale diverso da 0)
   - Esponente base 10
   - Separate dal simbolo “e”

   P.es., il numero 1853.0000059 può essere rappresentato come:

   185300.00059e-2 → $185300.00059*10^{-2}$
   18530000059e-7
   0.001853000059e+6
   **0.18530000059e+4** (*notazione standard*)

# Numeri Reali (*cont.*)

- Il numero di cifre disponibili per rappresentare un numero all'interno del computer è limitato, per cui si potrebbero verificare approssimazioni (*troncamenti* o *arrotondamenti*) nella rappresentazione di numeri reali 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.

# Tipi “float” e “double”

- **Mirano a rappresentare (*con diversa precisione*) un sottoinsieme dei numeri REALI**

- In realtà 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

# 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 |

# Operatori su “float” e “double”

| | |
|---|---|
| = | Assegnamento |
| + | Addizione |
| * | Moltiplicazione |
| / | Divisione reale (è diversa dalla divisione tra interi) |
| == | Uguaglianza |
| != | Diverso (*not equal*) |
| < | Minore |
| > | Maggiore |
| <= | Minore o uguale |
| >= | Maggiore o uguale |

# 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

- 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

# Operatori su “float” e “double”

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

# 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 (**%**)
- Operatore di assegnamento (**=**) e operatore di uguaglianza (**==**)
- Notazione prefissa e postfissa di **++** e **- -** negli assegnamenti

# Riassunto

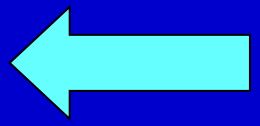
- Valori logici
- Valutazione in corto-circuito

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

- Tipo float e double

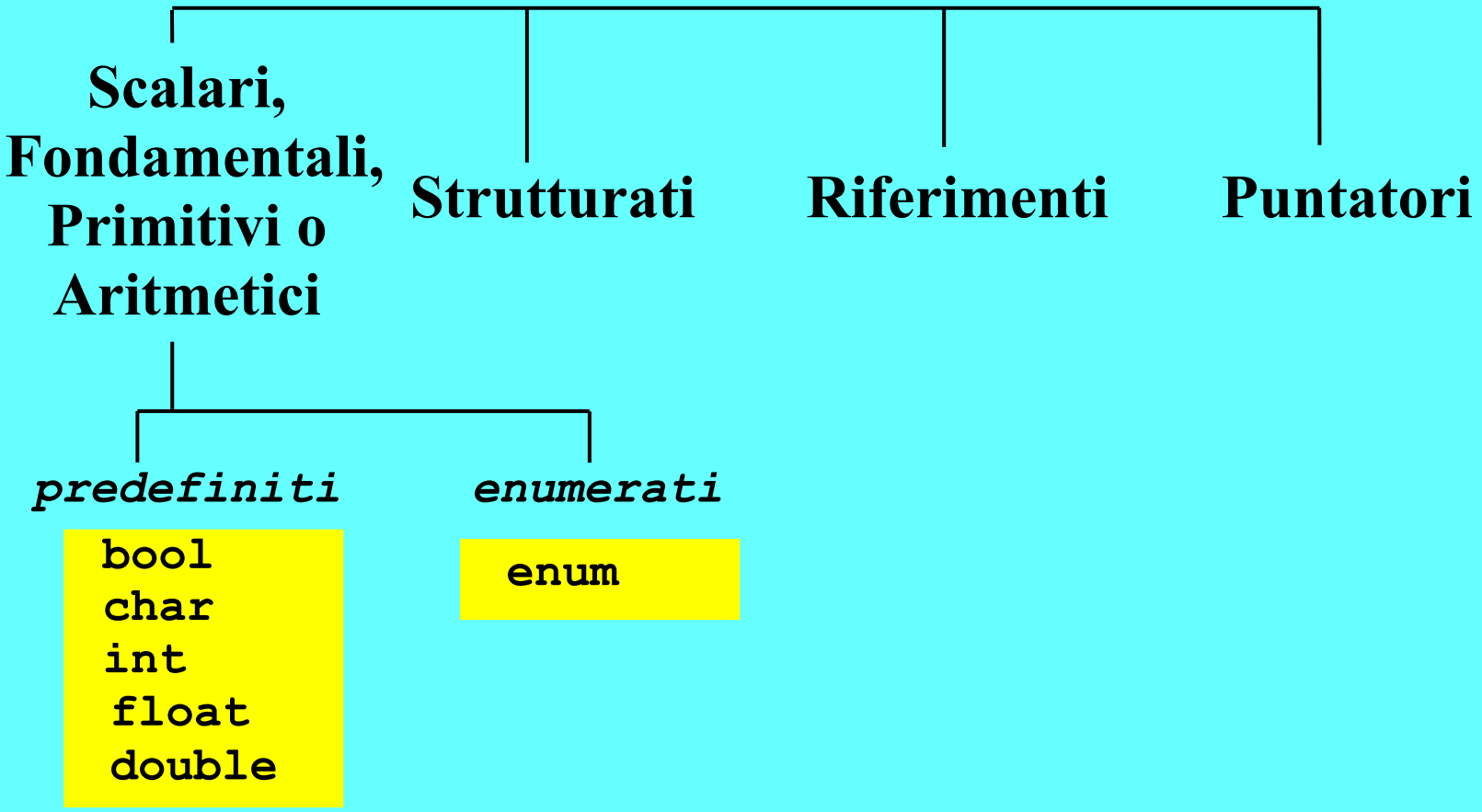
# Tipi di dato primitivi

- **int** (*già trattati*)
- **Valori logici**
- **Caratteri**
- **Reali**
- **Enumerati**



- **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

- **Insieme di costanti** definito dal programmatore
  - ciascuna individuata da un identificatore e detta **enumeratore**
- Esempio di dichiarazione:
  enum colori_t {rosso, verde, 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]

# Tipo enumerato

- 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 ;*

# Tipo enumerato

- 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 ;*

# Occupazione di memoria e range

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

  **Range (*logico*!) limitato all’elenco dei valori.**
- Ma non c’è controllo sul range 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 ed 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 posso 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

# Benefici del Tipo Enumerato

١) Decisamente migliore leggibilità

٢) Indipendenza dai valori esatti delle costanti

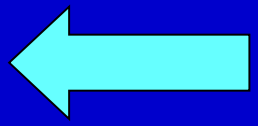
- Conseguenza importantissima: se cambio il valore di una costante, non devo modificare il resto del programma

٣) 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**
- **Reali**
- **Enumerati**
- **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)**

# 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 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"**

- 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
  - al limite, vengono 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, l'effettiva operazione effettuata sarà quella relativa all'operando con più alto livello gerarchico.**
   - Ogni variabile "char" o "short" viene convertita **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 è dello stesso tipo.

# **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 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 ovvero 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+a) e poi risulta un assegnamento omogeneo*/

d = c;      /* char → int → double      w==75. */

i = d;      /* sicuro troncamento di informazione della
               parte decimale (a==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 = 123456;
short varshort = static_cast<short> (varlong);
                                    /* Valore senza senso! */
```

(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!