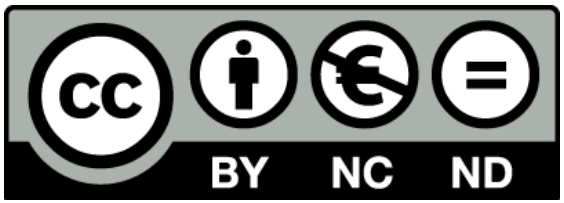
Laboratorio di Programmazione di Sistema

Conversione dei Dati

Luca Forlizzi, Ph.D.

Versione 23.1



Conversione dei Dati in *ASM*

- Molti linguaggi ad alto livello, come il C, offrono operatori che possono essere applicati ad operandi di svariati tipi
- Al contrario, le istruzioni *ASM* hanno dei vincoli più stretti sui formati di dato ammissibili per ciascuno dei propri operandi
- La maggior parte delle istruzioni *ASM* prevede che tutti gli operandi abbiano lo stesso formato di dato e usa per tutti gli operandi la stessa interpretazione di dato
- Di solito è il programmatore che deve fare in modo che i dati cui accedono determinate istruzioni abbiano lo stesso formato, scrivendo, quando necessario, del codice per convertire dati da un formato all'altro

Conversione dei Dati in *ASM*

- In alcune circostanze, durante l'esecuzione di istruzioni aritmetico/logiche o di trasferimento dati, si verificano delle conversioni di dato in modo automatico
- Altre conversioni vengono effettuate da apposite istruzioni
- Le conversioni in cui uno o entrambi i dati coinvolti sono interpretati come rappresentazioni di numeri floating-point avvengono, di solito, a seguito di apposite istruzioni e non in modo automatico; comunque, non verranno trattate in LPS, come tutto ciò che riguarda dati floating-point

Conversione dei Dati in *ASM*

- Nel seguito discutiamo alcuni procedimenti che vengono impiegati nelle conversioni tra formati interi e alcune delle circostanze in cui vengono usati
- Incontreremo altre applicazioni di tali procedimenti in future presentazioni
- Nell'illustrazione dei procedimenti di conversione
  - str_S è una stringa binaria contenuta in una parola che ha formato S di lunghezza $\mathtt{LEN}_S$
  - val_S è il valore rappresentato da str_S, in base ad una determinata interpretazione di dato
  - str_D è la stringa binaria prodotta a partire da str_S da un procedimento di conversione verso il formato D di lunghezza $\mathtt{LEN}_D$
  - val_D è il valore rappresentato da str_D, in base ad una determinata interpretazione di dato

Narrowing Conversion

- Si ha una *narrowing conversion* quando $\texttt{LEN}_S > \texttt{LEN}_D$
- Il procedimento di *narrowing conversion* più usato è chiamato **modulo-narrowing** e prevede di "estrarre" le cifre meno significative del dato da convertire
- La stringa str_D è costituita dalle $\texttt{LEN}_D$ cifre meno significative di str_S; le restanti cifre di str_S non vengono utilizzate per formare str_D
- Questo procedimento è motivato dal fatto che, con tutte le più comuni interpretazioni di dato per interi, se un valore v può essere rappresentato in modo corretto sia con $\texttt{LEN}_D$ che con $\texttt{LEN}_S$ cifre, la rappresentazione binaria di v con $\texttt{LEN}_D$ cifre è identica alla stringa ottenuta prendendo le $\texttt{LEN}_D$ cifre meno significative della rappresentazione binaria di v con $\texttt{LEN}_S$ cifre

Narrowing Conversion

- L'effetto del procedimento **modulo-narrowing** sui valori rappresentati dipende dalle interpretazioni di dato
  - Se entrambe le stringhe str_S e str_D vengono interpretate in codifica naturale si ha $\mathrm{val}_D = \mathrm{val}_S \mod 2^{\mathtt{LEN}_D}$, che implica, in particolare, che se val_S è rappresentabile in modo esatto con $\mathtt{LEN}_D$ cifre (ovvero $\mathrm{val}_S < 2^{\mathtt{LEN}_D}$), si ha $\mathrm{val}_D = \mathrm{val}_S$
  - Se entrambe le stringhe str_S e str_D vengono interpretate in complemento a 2 o complemento a 1
    - Se val_S è rappresentabile in modo esatto con $\mathtt{LEN}_D$ cifre, si ha $\mathrm{val}_D = \mathrm{val}_S$
    - Altrimenti può accadere che val_D abbia segno opposto a quello di val_S, e quindi non può essere definita una relazione esatta tra val_S e val_D: in questo caso quindi **modulo-narrowing** altera in maniera non definita il valore del dato convertito

Extending Conversion

- Si ha una *extending conversion* quando $\texttt{LEN}_S < \texttt{LEN}_D$
- In tali procedimenti, è necessario aggiungere cifre binarie a str_S per ottenere una stringa di $\texttt{LEN}_D$ cifre
- Ci sono due diversi procedimenti comunemente utilizzati ed entrambi prevedono che str_D sia formata aggiungendo a str_S, $\texttt{LEN}_D - \texttt{LEN}_S$ cifre nelle posizioni più significative

Extending Conversion

- Nel procedimento **zero-extension**, str_D viene così formata
  - le $\mathtt{LEN}_S$ cifre meno significative di str_D, sono uguali alle corrispondenti cifre di str_S
  - le restanti $\mathtt{LEN}_D - \mathtt{LEN}_S$ cifre vengono poste al valore 0
- **Zero-extension** preserva la semantica degli interi senza segno: infatti, se entrambe le stringhe str_S e str_D vengono interpretate in codifica naturale, si ha $\mathrm{val}_D = \mathrm{val}_S$

Extending Conversion

- Nel procedimento **sign-extension**, str_D viene così formata
  - le $\mathtt{LEN}_S$ cifre meno significative di str_D, sono uguali alle corrispondenti cifre di str_S
  - le restanti $\mathtt{LEN}_D - \mathtt{LEN}_S$ cifre vengono poste allo stesso valore della cifra più significativa di str_S
- **Sign-extension** preserva la semantica delle codifiche in complemento a 2 e complemento a 1, infatti, se entrambe le stringhe str_S e str_D vengono interpretate con una di tali codifiche, si ha $\mathrm{val}_D = \mathrm{val}_S$

Alcune Conversioni di Dato in MIPS32

- In MIPS32 le operazioni aritmetico-logiche su numeri interi vengono sempre effettuate tra dati di 32 cifre binarie
- Gli operandi in un'istruzione aritmetico-logica possono essere registri o operandi immediati
- L'unico formato disponibile per i registri ha 32 bit, ma gli operandi immediati possono avere 32 bit (formato `word`) oppure 16 bit (formato `half`)
- Se un'istruzione aritmetico-logica ha un operando immediato in formato `half` e gli altri operandi in formato `word`, il dato dell'operando in formato `half` viene convertito in modo automatico al formato `word`, prima di effettuare l'operazione specificata dall'istruzione

Alcune Conversioni di Dato in MIPS32

- In particolare, tra le istruzioni che abbiamo incontrato nelle precedenti lezioni, questa circostanza si verifica con le istruzioni `add`, `addu`, `sub` e `subu`
- Per tali istruzioni i primi due operandi sono registri ed hanno quindi formato `word` ma è possibile che Il terzo operando sia un operando immediato in formato `half`: in tal caso il contenuto del terzo operando viene automaticamente convertito al formato `word`, tramite **sign-extension**, prima di effettuare l'operazione specificata dall'istruzione
- Altre conversioni di dato automatiche si verificano con differenti istruzioni aritmetico-logiche e con istruzioni di trasferimento di dati in memoria che discuteremo in future presentazioni

Alcune Conversioni di Dato in MC68000

- I registri indirizzi, essendo orientati a contenere indirizzi di memoria, che in MC68000 sono valori di 32 bit, hanno delle regola particolari sui formati
- Innanzitutto, il formato `byte` non è valido per i registri indirizzi, quindi qualunque istruzione che abbia come operando almeno un registro indirizzi non può processare dati in formato `byte`
- In particolare, un'istruzione con estensione `.b` non può avere un registro indirizzi come operando

Alcune Conversioni di Dato in MC68000

- Il formato `word` (16 bit) è valido per i registri indirizzi, ma viene usato solo quando il registro indirizzi
  - non viene modificato dall'istruzione
  - non è il secondo operando dell'istruzione `cmp`
- Se un'istruzione modifica un registro indirizzi, oppure è `cmp` con un registro indirizzi come secondo operando, effettua la sua operazione tra dati di 32 bit
- Ciò accade anche quando l'istruzione ha estensione `.w`
- Se l'altro operando di una tale istruzione ha formato `word`, una copia del contenuto di tale operando viene automaticamente convertita dal formato `word` al formato `long`, tramite **sign-extension**, e usata per eseguire l'operazione specificata dall'istruzione

Alcune Conversioni di Dato in MC68000

- La conversione automatica di un operando dal formato `word` al formato `long` avviene quando l'operando destinazione (o il secondo operando di `cmp`) è un registro indirizzi, ma non se esso è un registro dati, come mostrato in Code 1

Code 1

```
move.l #0,d0       ; d0 contiene $00000000
move.l #0,a0       ; a0 contiene $00000000
move.w #$F000,d0   ; d0 contiene $0000F000
move.w #$F000,a0   ; a0 contiene $FFFFF000
```

- Altre conversioni di dato automatiche si verificano in differenti circostanze che studieremo in future presentazioni

Alcune Conversioni di Dato in MC68000

- Oltre alle conversioni automatiche, MC68000 supporta conversioni di dato sotto controllo del programmatore
- Le conversioni tramite **modulo-narrowing** e **zero-extension** sono supportate in modo "naturale" dalla capacità di effettuare operazioni arimtetico-logiche e di trasferimento dati con formati differenti
- Infatti, copiare da un dato memorizzato in una parola di registro, solo i bit contenuti in una parola di lunghezza minore dello stesso registro, equivale a effettuare un'operazione di **modulo-narrowing**

Alcune Conversioni di Dato in MC68000

- Code 2 mostra un esempio di **modulo-narrowing** mediante trasferimento dati da formato `word` a formato `byte`

Code 2

```
move.w #$1234,d1  ; la word di d1 contiene
                  ; il dato originale $1234
move.b d1,d2      ; il byte di d2 contiene
                  ; dato convertito $34
```

Alcune Conversioni di Dato in MC68000

- Copiare un dato memorizzato in una parola di registro, in una parola di lunghezza maggiore che ha tutti i bit al valore 0, equivale a effettuare un'operazione di **zero-extension**
- Code 3 mostra un esempio di **zero-extension** mediante trasferimento dati da formato `word` a formato `long`

Code 3

```
move.w #$1234,d1  ; la word di d1 contiene
                  ; il dato originale $1234
clr.l  d2         ; la long di d2
                  ; contiene $00000000
move.w d1,d2      ; la long di d2 contiene il
                  ; dato convertito $00001234
```

Alcune Conversioni di Dato in MC68000

- MC68000 fornisce l'istruzione `ext` per effettuare conversioni tramite **sign-extension** di parole contenute in un registro dati
- L'unico operando di `ext`
  - deve essere un registro dati
  - il suo formato è quello indicato dall'estensione, se presente, oppure `word` in caso contrario
- `ext.w` converte nel formato `word` il contenuto del `byte` dell'operando e copia il valore ottenuto nello stesso registro; ovvero i bit di posizione compresa tra 8 e 15 dell'operando vengono sovrascritti con il valore del bit che ha posizione 7
- `ext.l` converte nel formato `long` il contenuto della `word` dell'operando e copia il valore ottenuto nello stesso registro; ovvero i bit di posizione compresa tra 16 e 31 dell'operando vengono sovrascritti con il valore del bit che ha posizione 15

Alcune Conversioni di Dato in MC68000

- MC68020 e le successive versioni di M68000 offrono anche l'istruzione `extb.l` che converte nel formato `long`, mediante **sign-extension**, il contenuto del `byte` di un registro dati e copia il valore ottenuto nello stesso registro; ovvero i bit di posizione compresa tra 8 e 31 del registro vengono sovrascritti con il valore del bit che ha posizione 7

Type-Checking

- Diversamente dai linguaggi *ASM*, i linguaggi ad alto livello come il C offrono operatori che possono essere applicati ad operandi di svariati tipi
- Ciò rende più comoda e naturale la programmazione
- Tuttavia non tutti gli operatori ammettono operandi di qualunque tipo

Type-Checking

- Vi sono delle regole che specificano, per ciascun operatore, quali sono i tipi ammissibili per gli operandi
- Nella maggior parte dei casi, tali regole sono regole sintattiche o constraint e quindi, in base a C Standard, un'implementazione è tenuta a produrre un messaggio diagnostico quando esse vengono violate
- In altre parole, la maggior parte dei casi di violazione di una regola di ammissibilità per il tipo di un operando, deve essere rilevata e segnalata al programmatore dalle implementazioni

Type-Checking

- L'insieme dei controlli effettuati per rilevare violazioni di regole sintattiche e constraint relativi ai tipi ammissibili per gli operandi, viene comunemente chiamato *type-checking* o, in italiano, *controllo dei tipi* (si noti che type-checking è un termine in uso comune ma che non viene usato da C Standard)
- Tipicamente, nelle implementazioni di C Standard il type-checking è statico, in quanto il linguaggio non prevede che i tipi associati variabili e funzioni possano cambiare durante l'esecuzione del programma
- Quindi le violazioni di regole sintattiche e constraint che stabiliscono quali sono i tipi ammissibili per gli operandi di ciascuno degli operatori del linguaggio, vengono rilevate e segnalate al programmatore durante la traduzione in forma eseguibile del programma

Type Conversion

- Vi sono moltissime situazioni in cui, durante l'esecuzione di un programma, un valore di un certo tipo deve essere convertito in un tipo diverso
- La ragione forse più importante è che molti operatori, sebbene accettino operandi di una grande varietà di tipi, eseguono in realtà l'operazione ad essi associata dopo aver trasformato gli operandi in valori di una quantità più ristretta di tipi
- Ciò accade in quanto C Standard è stato definito in modo da consentire l'esecuzione efficiente delle operazioni e, nella maggior parte delle *ISA*, le operazioni vengono effettuate tra operandi che hanno lo stesso *formato*

Type Conversion

- Pertanto può accadere che un'implementazione di C Standard debba *convertire* alcuni degli operandi ad un diverso tipo, prima di effettuare l'operazione specificata dall'operatore
- Ad esempio, se in un'espressione vengono sommati un valore `short int` con uno di tipo `int`, un'implementazione in cui gli oggetti di `short int` sono parole di formato diverso rispetto a quelle che costituiscono gli oggetti di `int`, deve fare in modo che il valore di tipo `short int` venga convertito nel tipo `int` prima di eseguire la somma
- Le regole che stabiliscono in che modo e in quali circostanze avvengono le conversioni di tipo, sono piuttosto complesse a causa del fatto che C Standard ha numerosi tipi e a causa della backward compatibility

Type Conversion

- In generale, una *conversione di tipo* è un'operazione che viene effettuata su un operando, costituito da un'espressione `E`
- Il tipo di `E` viene detto *tipo sorgente*
- In base al tipo sorgente e al valore di `E`, la conversione di tipo produce un valore di un altro tipo, detto *tipo destinazione*

Type Conversion

- La relazione che lega il valore di `E` e il valore prodotto dalla conversione è stabilita dalle regole di conversione
- Nel caso di conversione tra 2 tipi aritmetici, il valore di `E` e il valore del risultato della conversione sono, di norma, lo stesso numero o 2 numeri diversi ma tra loro "vicini"
- Una conversione di tipo è un'operazione che non ha side-effects, ovvero non modifica in alcun modo `E`
- In particolare, se `E` denota una variabile, la conversione non modifica né il tipo associato a tale variabile, né il valore in essa memorizzato

Type Conversion

- È opportuno insistere su questo punto, in quanto può essere oggetto di fraintendimenti
- Spesso, descrivendo un programma in modo informale, si usano frasi come "la variabile `x` viene convertita a `double`"
- Un'interlocutore che non conosca bene il C, potrebbe interpretare una frase del genere in modo scorretto, pensando che vengano cambiati tipo e/o valore della variabile `x`
- Il vero significato della frase, invece, è il seguente
  - viene calcolato il valore di tipo `double` "più vicino possibile" (se possibile uguale) al valore memorizzato in `x`
  - tale valore è il risultato della conversione e viene utilizzato nella valutazione dell'espressione che contiene al suo interno la conversione di tipo
  - il valore memorizzato in `x` e il tipo associato a tale variabile, non vengono modificati

Type Conversion

- Il motivo per cui spesso ci si esprime con frasi potenzialmente ingannevoli, è che chi conosce ed è abituato al C, le interpreta sicuramente in modo corretto
- Infatti in C, come già detto, la *tipizzazione*, ovvero l'associazione tra una variabile e il suo tipo, è statica: viene stabilita durante la traduzione e non può essere in alcun modo modificata durante l'esecuzione del programma
- Altri linguaggi con tipizzazione statica sono in Java, Pascal, C++, mentre Python, Javascript e PHP sono esempi di linguaggi a tipizzazione dinamica

Type Conversion

- In molte situazioni, ad esempio quando gli operandi di un operatore aritmetico hanno tipi diversi, le conversioni di tipo avvengono automaticamente, senza che il programmatore debba segnalare la necessità della conversione scrivendo comandi appositi nel programma; per questo motivo tali conversioni sono dette *implicite*
- C Standard fornisce inoltre la possibilità di effettuare conversioni *esplicite*, tramite un operatore (chiamato operatore di *cast*) che viene scritto dal programmatore
- Nel caso di conversioni implicite, quale sia il tipo destinazione viene stabilito in base alle regole che verranno illustrate in seguito
- Nel caso di conversioni esplicite il tipo destinazione è specificato in modo esplicito per mezzo della sintassi

Regole sulle Conversioni di Tipo

- Sfortunatamente, le regole sulle conversioni di tipo in C Standard sono complesse, in particolare quelle relative alle conversioni implicite, perché
  - C Standard definisce un numero elevato di tipi aritmetici
  - L'esigenza di backward compatibility impone regole adatte a un gran numero di casi
- Pertanto, in ambito professionale, ogni qualvolta si scriva codice che contiene conversioni di tipo implicite è buona norma verificarne la correttezza facendo riferimento a **[C99]**, sopratutto quando si intendono realizzare in C applicazioni destinate a svariati ambienti operativi, con caratteristiche hardware diverse

Regole sulle Conversioni di Tipo

- Le regole di conversione implicita, si applicano alle seguenti situazioni
  1. Espressioni aritmetiche o logiche in cui sono presenti operandi di tipi diversi
  2. Assegnamenti in cui l'operando destro ha tipo diverso dall'operando sinistro
  3. Passaggi di parametro, in cui argomento e parametro hanno tipi diversi
  4. Istruzioni `return` seguite da un'espressione che ha tipo diverso da quello del risultato della funzione in cui si trova l'istruzione
- Le ultime 2 situazioni verranno esaminate nelle presentazioni dedicate allo studio delle funzioni

Conversioni Implicite nelle Espressioni Aritmetico-Logiche

- Nelle espressioni aritmetiche e logiche, viene effettuato il *type-checking* per verificare che sia possibile applicare gli operandi agli operatori che compongono le espressioni
- In caso positivo si procede alla valutazione, altrimenti si ha una constraint violation
- Nella valutazione degli operatori binari che hanno operandi di tipi diversi, prima di calcolare il risultato i valori degli operandi vengono convertiti ad uno stesso tipo mediante un insieme di regole denominato Usual Arithmetic Conversions (**UAC**)

Conversioni Implicite nelle Espressioni Aritmetico-Logiche

- Dati un valore v_1 di tipo T_1 e un valore v_2 di tipo T_2 che sono operandi di uno stesso operatore, la strategia generale di **UAC** è quella di convertire entrambi i valori al "più piccolo" tipo T che possa rappresentare tutti i valori sia di T_1 che di T_2
- Nella maggior parte dei casi, accade che T_2 può rappresentare tutti i valori di T_1 e quindi v_1 viene convertito a T_2 oppure viceversa che T_1 può rappresentare tutti i valori di T_2 e quindi v_2 viene convertito a T_1
- Le suddette conversioni avvengono mediante delle regole di *conversione diretta* di valori da un tipo S a un tipo D, descritte nel seguito
- I dettagli esatti di **UAC** sono piuttosto complessi, a causa delle numerose combinazioni di tipi che possono presentarsi

Conversioni Implicite negli Assegnamenti

- Nelle espressioni di assegnamento, viene effettuato il *type-checking* per verificare che sia possibile assegnare all'operando sinistro un valore del tipo dell'espressione che compare come operando destro
- Se l'assegnamento è possibile, il valore dell'operando destro viene convertito al tipo dell'operando sinistro mediante una *conversione diretta* dal tipo dell'operando destro al tipo dell'operando sinistro
- Il risultato dell'espressione di assegnamento ha il tipo dell'operando sinistro e il valore prodotto dalla conversione a tale tipo del valore dell'operando destro; come side-effect tale risultato viene memorizzato nell'operando sinistro
- Se l'operando sinistro e l'operando destro hanno entrambi un tipo aritmetico, l'assegnamento è sempre possibile

Conversioni Dirette

- Dati due tipi S (*tipo sorgente*) e D (*tipo destinazione*) C Standard stabilisce il comportamento di una abstract machine che effettua la conversione di un valore di tipo S nel tipo D attraverso delle regole dette di *conversione diretta*
- In questa presentazione descriviamo i principi generali e alcuni dettagli delle regole di conversione diretta tra tipi aritmetici
- Non verranno trattati i dettagli relativi alle regole che coinvolgono tipi di dato *complex* in quanto tali tipi non vengono trattati in LPS

Conversioni Dirette

- Le conversioni dirette tra tipi aritmetici seguono il seguente principio: se un valore v è rappresentabile sia in un tipo *sorgente* S che in un tipo *destinazione* D, la conversione di v da S a D non deve modificare v
- Si noti che se v è rappresentabile sia in S che in D, può accadere che le due stringhe binarie che rappresentano v, rispettivamente in S e in D, possono essere diverse e quindi l'operazione di conversione deve svolgere il compito di produrre una nuova stringa binaria
- Ad esempio il valore 10 nel tipo `float` viene rappresentato da una certa stringa $s1$; se esso viene convertito al tipo `int`, l'operazione di conversione deve produrre una stringa $s2$, che in molte implementazioni è diversa da $s1$ (ad esempio quelle che rappresentano `float` mediante codifica IEEE-754 e `int` mediante codifica in complemento a 2 con 32 cifre binarie)

Conversioni Dirette

- Conversioni dirette da un qualunque tipo aritmetico a `_Bool`
  - se il valore da convertire è pari a 0, il risultato della conversione è 0
  - altrimenti il risultato della conversione è 1
- Conversioni dirette da un qualunque tipo intero S a un tipo intero D senza segno diverso da `_Bool`
  - se il valore v da convertire è rappresentabile in D, il risultato della conversione è v
  - altrimenti il risultato della conversione è $v \mod K_D$, dove K_D è pari al massimo valore rappresentabile in D aumentato di 1

Conversioni Dirette

- Conversioni dirette da un qualunque tipo intero S a un tipo intero D con segno
  - se il valore v da convertire è rappresentabile in D, il risultato della conversione è v
  - altrimenti si ha un implementation defined behavior

Conversioni Dirette

- Conversioni dirette da un tipo real floating S a un tipo intero D diverso da `_Bool`
  - per prima cosa si ottiene un valore intero v' azzerando la parte frazionaria di v
  - se v' è rappresentabile in D, il risultato della conversione è v'
  - altrimenti si ha un undefined behavior

Conversioni Dirette

- Conversioni dirette da un tipo aritmetico S a un tipo real floating D
  - se il valore v da convertire è rappresentabile in D, il risultato della conversione è v
  - se v è compreso tra il più piccolo valore rappresentabile in D e il più grande valore rappresentabile in D, il risultato della conversione è un'approssimazione di v; in particolare è, a scelta dell'implementazione, uno tra i seguenti due valori
    - il più grande tra tutti i valori rappresentabili in D che sono minori di v
    - il più piccolo tra tutti i valori rappresentabili in D che sono maggiori di v
  - altrimenti si ha un undefined behavior

Conversioni Esplicite

- L'operatore di cast consente di specificare *in modo esplicito* un'operazione di conversione di tipo da effettuare su una espressione
- Un'espressione di cast ha la sintassi `( TIPO_DEST ) E`, dove `TIPO_DEST` è il nome di un tipo ed `E` è un'espressione
- In ogni espressione di cast viene effettuato il *type-checking* per controllare la compatibilità del tipo di `E` con `TIPO_DEST`
- In caso positivo, il valore di `E` viene convertito mediante la regola di conversione diretta dal tipo di `E` al tipo `TIPO_DEST`
- L'operatore di cast è un operatore unario ed ha precedenza maggiore rispetto agli operandi aritmetici binari
- Ad esempio `(float) dividend / divisor` è equivalente a `( (float) dividend ) / divisor`

Ulteriori Informazioni su Conversione dei Dati in C

- Il capitolo 7 di **[Ki]** fornisce ulteriori informazioni sulle regole di conversione dei dati in C, che è necessario conoscere per gli scopi di LPS e che pertanto sono rimandate allo studio autonomo
- Per approfondire ulteriormente, in prospettiva di un utilizzo professionale di C Standard, si consiglia la lettura attenta di **[C99]**