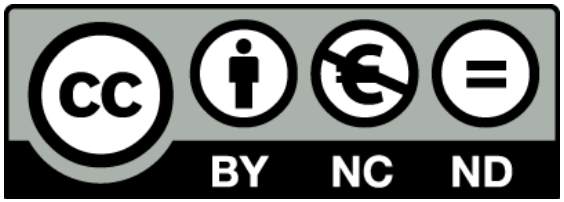
Laboratorio di Programmazione di Sistema

Conversione dei Dati

Luca Forlizzi, Ph.D.

Versione 20.3



Luca Forlizzi, 2020

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 in ingresso a 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, si verificano delle conversioni di dato in modo automatico, durante l'esecuzione di istruzioni di trasferimento dati o di istruzioni aritmetico/logiche
- 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, in maggioranza, 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, entrambi prevedono che str_D sia formata aggiungendo $\texttt{LEN}_D - \texttt{LEN}_S$ cifre a str_S 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
- Come sappiamo, 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`)
- Quando un'istruzione aritmetico-logica ha un operando immediato in formato `half`, legge il dato dall'operando e lo converte automaticamente al formato `word`, prima di effettuare l'operazione ad essa associata

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`
- Il terzo operando di tali istruzioni può essere un operando immediato in formato `half`, e in tal caso il suo contenuto viene automaticamente convertito al formato `word`, tramite **sign-extension**
- Altre conversioni di dato automatiche si verificano con differenti istruzioni aritmetico-logiche e con istruzioni di trasferimento di dati in memoria che studieremo 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`
- 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`

Alcune Conversioni di Dato in MC68000

- 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
- Quindi, se l'altro operando ha formato `word`, il suo contenuto viene automaticamente convertito al formato `long`, tramite **sign-extension**
- 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

Alcune Conversioni di Dato in MC68000

- 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**
- Code 1 mostra un esempio di **modulo-narrowing** mediante trasferimento dati da formato `word` a formato `byte`

Code 1

```
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 2 mostra un esempio di **zero-extension** mediante trasferimento dati da formato `word` a formato `long`

Code 2

```
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 il contenuto del `byte` dell'operando nel formato `word` 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 il contenuto della `word` dell'operando nel formato `long` 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, mediante **sign-extension**, il contenuto del `byte` di un registro dati nel formato `long` 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, un'implementazione è tenuta a rilevare e segnalare al programmatore, la maggior parte dei casi di violazione di una regola di ammissibilità per il tipo di un operando

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)
- Nella maggior parte delle implementazioni C, il type-checking è statico
- Ovvero le violazioni alle regole sintattiche e ai 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 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 trasformare i tipi di alcuni degli operandi, 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 che memorizza valori `short int` in 16 bit e valori `int` in 32 bit, deve fare in modo che il valore di tipo `short int` venga convertito nel tipo `int` prima di eseguire la somma

Type Conversion

- Tali conversioni 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
- 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

- Iniziamo con il precisare il concetto di conversione di tipo
- 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*
- 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

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" (al limite 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 in Java, Pascal, C++, 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, come Javascript o PHP, consentono invece di modificare il tipo associato ad una variabile durante l'esecuzione, e vengono pertanto detti linguaggi a tipizzazione dinamica

Regole di Conversione di Tipo

- Ci sono 2 categorie di regole per le conversioni di tipo
  1. Regole che definiscono come avvengono le conversioni, suddivise in
     - regole di conversione diretta
     - regole di conversione aggregate
  2. Regole che definiscono in quali circostanze avvengono le conversioni, suddivise in
     - regole di conversione implicita
     - regole di conversione esplicita

Regole di Conversione di Tipo

- In questa presentazione descriviamo le regole di conversione tra tipi base, con l'esclusione di
  - regole che riguardano i tipi floating complex, che sono tipi che rappresentano numeri complessi (presenti in C99 e nelle versioni successive)
  - regole che riguardano i tipi interi estesi, che sono dei tipi interi aggiuntivi a quelli standard che possono essere presenti in alcune implementazioni di C99 o di versioni successive
- Le conversioni tra tipi base sono sempre possibili in base alle regole del type-checking (in future presentazioni incontreremo conversioni che invece non sono possibili, nelle quali almeno uno dei tipi coinvolti non è un tipo base)

Regole di Conversione di Tipo

- Ogni tipo T rappresenta un insieme di valori $\mathrm{Vals}(T)$ appartenenti ad un certo insieme, attraverso un insieme di stringhe binarie $\mathrm{Strs}(T)$, tutte di uguale lunghezza $\mathrm{Len}(T)$
- Più esattamente, ogni valore $v \in \mathrm{Vals}(T)$ è rappresentato da una o più stringhe $s \in \mathrm{Strs}(T)$
- Nell'illustrazione delle regole di conversione da un tipo sorgente a un tipo destinazione
  - Il tipo sorgente viene indicato con S
  - Per ogni valore $v \in \mathrm{Vals}(S)$, $\mathrm{str}_S(v)$ indica la rappresentazione di v in S
  - Il tipo destinazione viene indicato con D
  - Per ogni valore $v \in \mathrm{Vals}(D)$, $\mathrm{str}_D(v)$ indica la rappresentazione di v in D

Regole di Conversione Diretta

- Per ciascun tipo di dato sorgente S e ciascun tipo di dato destinazione D, la *regola di conversione diretta* da S a D è una funzione da $\mathrm{Vals}(S)$ a $\mathrm{Vals}(D)$ che associa a un valore $v_S \in \mathrm{Vals}(S)$ un corrispondente valore $v_D \in \mathrm{Vals}(D)$
- Una regola di conversione diretta può essere costituita da una funzione parziale; in altre parole, possono esistere valori $v_S \in \mathrm{Vals}(S)$ per i quali la conversione nel tipo D non è definita

Regole di Conversione Diretta

- I valori rappresentati da un tipo aritmetico sono numeri
- Infatti ciascun tipo aritmetico viene definito per poter rappresentare un sottoinsieme (finito) di uno dei principali insiemi numerici usati in matematica
- Per ciascun tipo aritmetico T chiamiamo *insieme di supporto* $\mathrm{Support}(T)$ l'insieme (infinito) di numeri che T intende rappresentare
  - Per ciascun tipo intero senza segno T, definiamo $\mathrm{Support}(T)$ come l'insieme dei numeri naturali **N**
  - Per ciascun tipo intero con segno T, definiamo $\mathrm{Support}(T)$ come l'insieme dei numeri interi **Z**
  - Per ciascun tipo real floating T (ovvero `float`, `double`, `long double`), definiamo $\mathrm{Support}(T)$ come l'insieme dei numeri reali **R**
- Per ciascun tipo aritmetico T si ha $\mathrm{Vals}(T) \subset \mathrm{Support}(T)$

Regole di Conversione Diretta

- Inoltre, per ogni tipo aritmetico T, definiamo *range* di T, indicato come $\mathrm{Range}(T)$, il sottoinsieme di $\mathrm{Support}(T)$, tale che ogni elemento di $\mathrm{Range}(T)$ è compreso tra il minimo e il massimo di $\mathrm{Vals}(T)$
- Per ogni tipo intero T, si ha $\mathrm{Range}(T) = \mathrm{Vals}(T)$
- Per i tipi real floating T ciò non è vero e in particolare può accedere che un valore appartenga a $\mathrm{Range}(T)$ ma non a $\mathrm{Vals}(T)$

Regole di Conversione Diretta

- Alla base di tutte le regole di conversione tra tipi aritmetici vi è il seguente principio

Principio-base delle conversioni tra tipi aritmetici

Se per un numero v si ha sia $v \in \mathrm{Vals}(S)$ che $v \in \mathrm{Vals}(D)$, allora la conversione non deve modificare v

- Si noti che sebbene v sia rappresentabile sia in S che in D, $\mathrm{str}_S(v)$ e $\mathrm{str}_D(v)$ possono essere stringhe diverse, quindi l'operazione di conversione deve comunque svolgere il compito di produrre $\mathrm{str}_D(v)$
- Ad esempio il valore 10 nel tipo `float` viene rappresentato da $\mathrm{str}_{\texttt{float}}(10)$; se esso viene convertito al tipo `int`, l'operazione di conversione deve produrre la stringa $\mathrm{str}_{\texttt{int}}(10)$, che in molte implementazioni (ad esempio quelle che rappresentano `float` mediante codifica IEEE-754) è diversa da $\mathrm{str}_{\texttt{float}}(10)$

Regole di Conversione Diretta

Conversione da un tipo S al tipo `_Bool`

Sia $v \in \mathrm{Vals}(S)$

- Se v è 0, la conversione produce $\mathrm{str}_{\texttt{\_Bool}}(0)$, ovvero la rappresentazione di 0 in `_Bool`
- Altrimenti, la conversione produce $\mathrm{str}_{\texttt{\_Bool}}(1)$

Regole di Conversione Diretta

Conversione da un tipo intero S a un tipo intero D senza segno diverso da `_Bool`

Sia $v \in \mathrm{Vals}(S)$

- Se $v \in \mathrm{Vals}(D)$, la conversione produce $\mathrm{str}_D(v)$
- Altrimenti, v viene convertito aggiungendo o togliendo ad esso, ripetutamente, il più grande valore rappresentabile in D aumentato di 1, fino a quando non si ottiene un valore $v' \in \mathrm{Vals}(D)$; la conversione quindi produce $\mathrm{str}_D(v')$

- In modo equivalente, se $|\mathrm{Vals}(D)|$ è la quantità di diversi valori rappresentabili da D, si ha $v' = v \mod |\mathrm{Vals}(D)|$

Regole di Conversione Diretta

- Nel caso in cui S è un tipo intero senza segno, oppure un tipo intero con segno rappresentato in complemento a 2, l'operazione di somma o sottrazione ripetuta del più grande valore rappresentabile in D aumentato di 1, menzionata nel secondo caso della regola, può essere effettuata in maniera efficiente da una semplice operazione che produce immediatamente la rappresentazione di v', ovvero $\mathrm{str}_D(v')$
  - Se $\mathrm{Len}(S) = \mathrm{Len}(D)$, si costruisce $\mathrm{str}_D(v')$ facendo una semplice copia di $\mathrm{str}_S(v)$
  - Se $\mathrm{Len}(S) > \mathrm{Len}(D)$, si costruisce $\mathrm{str}_D(v')$ facendo **modulo-narrowing** di $\mathrm{str}_S(v)$ per eliminare le cifre in eccesso
  - Se $\mathrm{Len}(S) < \mathrm{Len}(D)$, si costruisce $\mathrm{str}_D(v')$ facendo **zero-extension** di $\mathrm{str}_S(v)$

Regole di Conversione Diretta

- Ad esempio supponiamo che S sia rappresentato da stringhe binarie di 16 cifre adottando la codifica in complemento a 2 e che D sia rappresentato con stringhe binarie di 8 cifre (dunque il valore massimo rappresentabile in D è 255)
  - la rappresentazione di -16 in S è $\mathrm{str}_S(-16) = 1111111111110000$
  - in base alla regola, il risultato della conversione deve essere $-16 + 256$, ovvero 240
  - la rappresentazione di 240 in D è $\mathrm{str}_S(240) = 11110000$, che è formata dalle 8 cifre meno significative di $\mathrm{str}_S(-16)$

Regole di Conversione Diretta

Conversione da un tipo intero S a un tipo intero D con segno

Sia $v \in \mathrm{Vals}(S)$

- Se $v \in \mathrm{Vals}(D)$, la conversione produce $\mathrm{str}_D(v)$
- Altrimenti, il risultato della conversione è un valore di D definito dall'implementazione oppure viene generata un'eccezione definita dall'implementazione

Regole di Conversione Diretta

Conversione da un tipo real floating S a un tipo intero D diverso da `_Bool`

Sia $v \in \mathrm{Vals}(S)$

- La parte frazionaria di v viene azzerata (ovvero si approssima verso 0 il valore v) ottenendo un valore intero v'
- Se $v' \in \mathrm{Vals}(D)$, la conversione produce $\mathrm{str}_D(v')$
- Altrimenti, si ha un undefined behavior

Regole di Conversione Diretta

Conversione da un tipo S a un tipo real floating D

Sia $v \in \mathrm{Vals}(S)$

- Se $v \in \mathrm{Vals}(D)$, la conversione produce $\mathrm{str}_D(v)$
- Altrimenti, se $v \in \mathrm{Range}(D)$, il risultato della conversione è il più grande tra i valori rappresentabili in D minori di v oppure il più piccolo tra i valori rappresentabili in D maggiori di v, a scelta dell'implementazione
- Altrimenti, si ha un undefined behavior

Regole di Conversione Aggregate

- Ad ogni tipo intero è associato un numero intero detto *integer conversion rank*
- I valori dei rank dei tipi standard sono definiti dalle implementazioni, ma, in base a C Standard, devono soddisfare le seguenti relazioni
  - Ogni rank di un tipo senza segno è uguale al rank del corrispondente tipo con segno
  - Il rank di `long long int` è maggiore del rank di `long int`
  - Il rank di `long int` è maggiore del rank di `int`
  - Il rank di `int` è maggiore del rank di `short int`
  - Il rank di `short int` è maggiore del rank di `signed char`
  - Il rank di `char` è uguale al rank di `signed char`
  - Il rank di `_Bool` è minore del rank di `char`

Regole di Conversione Aggregate

- Nel seguito presentiamo alcune tra le principali regole di conversione aggregate
  - Integer Promotions (**IP**)
  - Usual Arithmetic Conversions (**UAC**)
- Altre regole di conversione aggregate chiamate Default Argument Promotions (**DAP**) verranno introdotte in occasione di un futuro approfondimento sulle funzioni

Integer Promotions

- Questa regola si applica, in determinate circostanze, a un valore v di un tipo intero T

IP

- Se T ha rank maggiore o uguale al rank di `int` la conversione non modifica né v né il tipo di v
- Altrimenti, se $\mathrm{Vals}(T) \subseteq \mathrm{Vals}(\texttt{int})$, v viene convertito ad `int` applicando la regola di conversione diretta da T ad `int`
- Altrimenti v viene convertito ad `unsigned int` applicando la regola di conversione diretta da T ad `unsigned int`

Integer Promotions

- In altre parole, la regola **IP** converte un valore v
  - dai tipi `_Bool`, `signed char`, `short` al tipo `int`
  - dal tipo `unsigned char` o `unsigned short`
    - al tipo `int` se quest'ultimo è in grado di rappresentare tutti i valori del tipo sorgente
    - al tipo `unsigned int` in caso contrario
- Ad esempio, in un'implementazione in cui
  - `signed char`, `unsigned char` sono rappresentati da stringhe binarie di 8 cifre
  - `signed short`, `unsigned short` sono rappresentati da stringhe binarie di 16 cifre
  - `signed int`, `unsigned int` sono rappresentati da stringhe binarie di 32 cifre

  la regola converte i valori di tipo `_Bool`, `signed char`, `unsigned char`, `short`, `unsigned short` al tipo `int` e lascia immutati valori di altro tipo

Integer Promotions

- In un'implementazione in cui
  - `signed char`, `unsigned char` sono rappresentati da stringhe binarie di 8 cifre
  - `signed short`, `unsigned short` sono rappresentati da stringhe binarie di 16 cifre
  - `signed int`, `unsigned int` sono rappresentati da stringhe binarie di 16 cifre

  la regola converte i valori di tipo `_Bool`, `signed char`, `unsigned char`, `short` al tipo `int`, quelli di tipo `unsigned short` al tipo `unsigned int` e lascia immutati valori di altro tipo
- La regola **IP** viene applicata dalle altre regole di conversione aggregate, come discusso nel seguito

Usual Arithmetic Conversions

- Questa regola si applica a una coppia di valori v_1 di tipo T_1 e v_2 di tipo T_2, con $T_1 \neq T_2$, allo scopo di ottenere 2 valori dello stesso tipo T
- Uno o entrambi i valori vengono convertiti
- La strategia generale è quella di convertire i 2 valori nel "più piccolo" tipo che possa rappresentare tutti i valori di T_1 e T_2, ovvero nel tipo T tale che
  - $\mathrm{Vals}(T_1) \subseteq \mathrm{Vals}(T)$
  - $\mathrm{Vals}(T_2) \subseteq \mathrm{Vals}(T)$
  - per ogni tipo S, se $\mathrm{Vals}(T_1) \subseteq \mathrm{Vals}(S)$ e $\mathrm{Vals}(T_2) \subseteq \mathrm{Vals}(S)$, allora $\mathrm{Vals}(T) \subseteq \mathrm{Vals}(S)$
- Nella maggior parte dei casi, si ha $\mathrm{Vals}(T_1) \subseteq \mathrm{Vals}(T_2)$, e quindi v_1 viene convertito a T_2, oppure $\mathrm{Vals}(T_2) \subseteq \mathrm{Vals}(T_1)$ e quindi v_2 viene convertito a T_1

Usual Arithmetic Conversions

- Le regole (del C99) per eseguire **UAC** sono divise in due casi
  1. Il tipo di almeno uno dei valori è un tipo floating
  2. Entrambi i valori hanno un tipo intero
- Descriviamo **UAC** in forma semplificata, in quanto, come detto in precedenza, non consideriamo i casi che coinvolgono tipi complex floating

Usual Arithmetic Conversions

UAC Caso 1: Il tipo di almeno uno dei valori è un tipo real floating

1. Se uno dei valori ha tipo `long double`, allora l'altro viene convertito dal suo tipo a `long double`, in base alla relativa regola di conversione diretta
2. Altrimenti, se uno dei valori ha tipo `double`, allora l'altro viene convertito dal suo tipo a `double`, in base alla relativa regola di conversione diretta
3. Altrimenti, uno dei valori ha tipo `float`, e l'altro viene convertito dal suo tipo a `float`, in base alla relativa regola di conversione diretta

Usual Arithmetic Conversions

UAC Caso 2: Entrambi i valori v_1, v_2 hanno un tipo intero

1. Ad entrambi i valori viene applicata **IP**; se i valori risultanti v_1', v_2' hanno lo stesso tipo, allora la conversione **UAC** è completa
2. Altrimenti se i tipi di v_1' e v_2' sono entrambi con segno o entrambi senza segno, si converte il valore il cui tipo ha rank minore, nell'altro tipo, in base alla relativa regola di conversione diretta

Usual Arithmetic Conversions

UAC Caso 2: Entrambi i valori v_1, v_2 hanno un tipo intero

3. Altrimenti, se il tipo senza segno ha rank maggiore o uguale a quello del tipo con segno, si converte il valore del tipo con segno nell'altro tipo, in base alla relativa regola di conversione diretta
4. Altrimenti, se il tipo con segno può rappresentare l'insieme dei valori rappresentabili dal tipo senza segno, si converte il valore del tipo senza segno nell'altro tipo, in base alla relativa regola di conversione diretta
5. Altrimenti, entrambi i valori vengono convertiti al tipo senza segno che ha lo stesso rank del tipo con segno, in base alle relative regole di conversione diretta

Conversioni Implicite

- Le regole di conversione implicita stabiliscono in quali situazioni viene effettuata una conversione di tipo in modo automatico
- Le conversioni implicite avvengono
  - Nelle espressioni di assegnamento
  - Nelle espressioni aritmetiche e logiche
  - Nelle chiamate di funzione
  - Nelle istruzioni `return`
- Le conversioni implicite nelle chiamate di funzione e nelle istruzioni `return` verranno introdotte in occasione di un futuro approfondimento sulle funzioni

Conversioni Implicite nell'Assegnamento

- 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 la regola di 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 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 aritmetici unari $+$ e $-$, se l'operando ha tipo intero, ad esso viene applicata **IP** prima di calcolare il risultato
- Nella valutazione degli operatori aritmetici binari $+$, $-$, $*$, $/$, %, ai valori degli operandi viene applicata **UAC** prima di calcolare il risultato

Conversioni Implicite nelle Espressioni Aritmetico-Logiche

- Nella valutazione degli operatori relazionali $==$, $!=$, $<$, $>$, $<=$, $>=$, se entrambi gli operandi hanno tipo aritmetico, ai loro valori viene applicata **UAC** prima di calcolare il risultato
- Nella valutazione dell'operatore condizionale si procede come segue
  1. Si applica **UAC** al secondo e al terzo operando, al solo scopo di determinare il tipo T del risultato (ovvero si usa **UAC** per calcolare il tipo del risultato, ma il secondo e il terzo operando non vengono modificati)
  2. Il valore del risultato è quello del secondo o del terzo operando convertito a T mediante la regola di conversione diretta appropriata

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`

Esempi di Conversioni di Tipo

Che operazioni effettua Code 3?

Code 3

```
int x;  float f = 12.90f;
x = (int)f * 5;
```

Esempi di Conversioni di Tipo

Che operazioni effettua Code 3?

Code 3

```
int x;  float f = 12.90f;
x = (int)f * 5;
```

- Converte il valore di `f` al tipo `int`, ottenendo un nuovo valore che moltiplica per 5; memorizza il prodotto ottenuto in `x`
- Dopo l'esecuzione
  - `x` vale 60
  - `f` vale 12.90
- Sottolineamo, ancora una volta, che il valore memorizzato in `x`, così come il tipo di `x` non vengono modificati

Esempi di Conversioni di Tipo

Che operazioni effettua Code 4?

Code 4

```
unsigned char uc = 192;  unsigned u;
u = uc;
```

Esempi di Conversioni di Tipo

Che operazioni effettua Code 4?

Code 4

```
unsigned char uc = 192;  unsigned u;
u = uc;
```

- Calcola il valore da assegnare a `u` tramite una conversione diretta da `unsigned char` a `unsigned`: poiché 192 è rappresentabile in entrambi i tipi, `u` assume tale valore
- Dopo l'esecuzione
  - `u` ha tipo `unsigned` e vale 192
  - `uc` ha tipo `unsigned char` e vale 192
- Sottolineiamo, ancora una volta, che il valore memorizzato in `uc`, così come il tipo di `uc` non vengono modificati

Esempi di Conversioni di Tipo

Che operazioni effettua Code 5?

Code 5

```
char c = -6;   int i;
i = c;
```

Esempi di Conversioni di Tipo

Che operazioni effettua Code 5?

Code 5

```
char c = -6;   int i;
i = c;
```

- Calcola il valore da assegnare a `i` tramite una conversione diretta da `char` a `int`: `i` assume il valore -6 in quanto esso è rappresentabile anche in `int`
- Dopo l'esecuzione
  - `i` ha tipo `int` e vale -6
  - `c` ha tipo `char` e vale -6
- Sottolineamo, ancora una volta, che il valore memorizzato in `c`, così come il tipo di `c` non vengono modificati

Esempi di Conversioni di Tipo

Che operazioni effettua Code 6?

Code 6

```
long x, y = 1000000;  int z = 10000;
x = y + 200*z;
```

Esempi di Conversioni di Tipo

Che operazioni effettua Code 6?

Code 6

```
long x, y = 1000000;  int z = 10000;
x = y + 200*z;
```

- Per prima cosa effettua la moltiplicazione, i cui operandi hanno tipo `int`; in base ad **UAC** esegue la moltiplicazione tra 2 valori `int` e il risultato della moltiplicazione è un `int`
- In implementazioni in cui `int` ha 16 bit, si ha un overflow
- Poi esegue la somma tra un valore `int` e uno `long`, quindi per **UAC** converte l'operando `int` in `long`
- Il risultato della somma ha tipo `long` e viene memorizzato in `x` che ha di tipo `long`

Esempi di Conversioni di Tipo

Che operazioni effettua Code 7?

Code 7

```
int i;  unsigned short limite = 10
for (i=-10;i<limite;i++) printf("%d", i);
```

Esempi di Conversioni di Tipo

Che operazioni effettua Code 7?

Code 7

```
int i;  unsigned short limite = 10
for (i=-10;i<limite;i++) printf("%d", i);
```

- Dipende dall'implementazione, in particolare dalla rappresentazione dei tipi usata
- L'espressione `i<limite` confronta un operando di tipo `unsigned short` con uno di tipo `int`, quindi si applica la regola **UAC**, la quale a sua volta applica la regola **IP** a entrambi gli operandi

Esempi di Conversioni di Tipo

Che operazioni effettua Code 7?

Code 7

```
int i;  unsigned short limite = 10
for (i=-10;i<limite;i++) printf("%d", i);
```

- Nelle implementazioni in cui `int` può rappresentare tutti i valori di `unsigned short`, la regola **IP** converte il valore di `limite` a `int`
- Il risultato di tale conversione è il valore 10, e quindi il ciclo esegue 20 iterazioni

Esempi di Conversioni di Tipo

Che operazioni effettua Code 7?

Code 7

```
int i;  unsigned short limite = 10
for (i=-10;i<limite;i++) printf("%d", i);
```

- Nelle altre implementazioni, la regola **IP** converte `limite` al tipo `unsigned int`
- Di conseguenza, poiché a questo punto il primo operando di < ha tipo `int` e l'altro ha tipo `unsigned int`, in base a **UAC**, anche `i` viene convertito ad `unsigned int`

Esempi di Conversioni di Tipo

Che operazioni effettua Code 7?

Code 7

```
int i;  unsigned short limite = 10
for (i=-10;i<limite;i++) printf("%d", i);
```

- Per la regola di conversione diretta da un tipo intero con segno a uno intero senza segno, poiché -10 non è rappresentabile in `unsigned int`, a tale valore si aggiunge ripetutamente il più grande valore rappresentabile in `unsigned int` aumentato di 1, fino ad ottenere un valore rappresentabile in `unsigned int`
- In molte implementazioni, il risultato è un valore molto grande e di conseguenza il ciclo non esegue alcuna iterazione

Esempi di Conversioni di Tipo

Che operazioni effettua Code 8?

Code 8

```
unsigned char uc1 = 255, uc2 = 10;
double d;
d = uc1 * uc2;
```

Esempi di Conversioni di Tipo

Che operazioni effettua Code 8?

Code 8

```
unsigned char uc1 = 255, uc2 = 10;
double d;
d = uc1 * uc2;
```

- A prima vista si potrebbe pensare che la moltiplicazione tra il valore di `uc1` e quello di `uc2` produca un overflow, in quanto 2550 non è rappresentabile, in molte implementazioni, nel tipo `unsigned char`
- Non è così: infatti, per **UAC**, prima di procedere al calcolo del prodotto i due operandi vengono entrambi convertiti a `int` o a `unsigned`, mediante **IP**

Esempi di Conversioni di Tipo

Che operazioni effettua Code 8?

Code 8

```
unsigned char uc1 = 255, uc2 = 10;
double d;
d = uc1 * uc2;
```

- Pertanto, anche nelle implementazioni in cui il valore massimo per il tipo `unsigned char` è minore di 2550, non vi è overflow
- C Standard infatti garantisce che il numero 2550 è rappresentabile sia nel tipo `int` che nel tipo `double`

Sessione di Esercizi: Conversione dei Dati

- Per studiare in dettaglio alcuni argomenti introdotti in questa presentazione si veda la sezione 7.4 di **[Ki]**
- Svolgere il gruppo di esercizi **Conversione dei Dati**
- Per ulteriori spiegazioni sui contenuti degli esercizi, si vedano **[C99]**, **[M68000]** e **[MIPS32]**