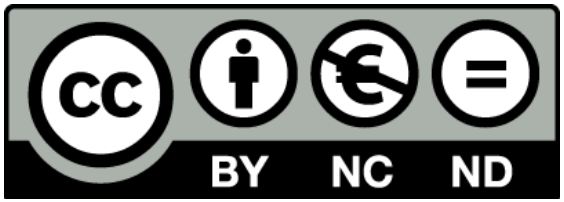
Laboratorio di Programmazione di Sistema

Organizzazione dei Dati in Memoria 3

Luca Forlizzi, Ph.D.

Versione 20.1



Organizzazione dei Dati in C

- In questa presentazione si discute di come si possano organizzare e gestire i dati contenuti nelle variabili di un programma C
- A tale scopo, ipotizziamo di tradurre il programma C nel linguaggio assembly definito da un *ASM-PM*, e indichiamo con *M* la *abstract machine* definita dall'*ASM-PM*
- Naturalmente, il modo di gestire un dato dipende dal tipo della variabile che lo contiene
- Poiché il numero di registri di *M* è limitato, mentre il numero di variabili di un programma C può essere molto grande, è chiaro che in generale è necessario immagazzinare i dati in memoria, in quanto i registri non sarebbero sufficienti

Organizzazione dei Dati in C

- Come sappiamo C Standard è stato definito in modo tale consentire implementazioni efficienti
- C Standard non definisce completamente le caratteristiche dei tipi aritmetici
- In particolare non stabilisce la quantità di bit usata dalle variabili di un determinato tipo, ma solo alcuni vincoli
- Ciò viene fatto deliberatamente, per consentire a ciascuna implementazione di organizzare le variabili di ciascun tipo nel modo che essa ritiene più opportuno
- Ovvero, tipicamente, nel modo più efficiente possibile date le caratteristiche dell'implementazione

Tipi di Dato Aritmetici

- Il modo più efficiente di gestire le variabili di un tipo aritmetico del C, è memorizzare, quando possibile, ciascuna delle variabili di un tipo, mediante parole di uno dei formati di dato di M che abbiano una dimensione appropriata
- Infatti, come sappiamo, le parole sono i gruppi di bit cui M può accedere con la massima efficienza

Tipi di Dato Aritmetici

- Ad esempio, in una traduzione nell'*ASM* MC68000, memorizzare le variabili di tipo `int` con gruppi di 24 bit non sarebbe una buona scelta, in quanto essa costringerebbe a tradurre ogni accesso ad una variabile di tipo `int` in uno dei seguenti modi, ciascuno dei quali non è molto efficiente, in termini di tempo o di spazio
  - accesso a 2 parole, un `byte` (8 bit) e una `word` (16 bit)
  - accesso ad una `long` (32 bit), seguito da operazioni di manipolazione dei bit che selezionino i soli 24 bit dedicati a memorizzare il dato
  - accesso ad una `long` (32 bit), lasciando inutilizzati gli 8 bit superflui di tale parola

Tipi di Dato Aritmetici

- Nella maggioranza delle traduzioni, quindi, si sceglie di memorizzare ogni dato che ha tipo aritmetico, in una singola parola e di rendere disponibili, per le variabili di ciascun tipo aritmetico, tutti i bit della parole usate per memorizzare tali variabili
- In altre parole, ad ogni tipo aritmetico **T** del C, viene associato un formato di dato **F** dell'*ASM-PM*, e si memorizza ciascun valore di **T** con una singola parola di **F**

Tipi di Dato Aritmetici

- Ricordiamo che, tra gli obiettivi di C Standard, c'è quello di ottenere un linguaggio paragonabile all'*ASM* non solo in termini di velocità di esecuzione, ma anche per quanto riguarda flessibilità ed efficienza nella gestione della memoria
- Uno dei motivi principali per cui C Standard fornisce un così elevato numero di tipi aritmetici, è per consentire di disporre in C di variabili associate a tutti i formati di dato disponibili nell'*ASM-PM* (in modo equivalente nella *ISA*) usata per la traduzione di un programma, in funzione dell'obiettivo sopra ricordato
- Quindi in genere le implementazioni di C Standard fanno in modo che ciascuno dei formati standard disponibili nell'*ASM* su cui la traduzione viene realizzata, venga associato ad almeno uno dei tipi di dato del C

Tipi di Dato Aritmetici

- I tipi `unsigned char` e `signed char` sono associati quasi sempre al formato *byte* di M, interpretato, rispettivamente, come intero senza segno e come intero con segno nella codifica utilizzata dall'implementazione
- Ricordiamo che C Standard ammette 3 possibili codifiche per interi con segno
  - modulo e segno
  - complemento a 1
  - complemento a 2

Tipi di Dato Aritmetici

- Come sappiamo `sizeof(char)` vale 1, mentre la dimensione degli altri tipi può essere maggiore
- Dunque le variabili di ciascun tipo intero hanno una quantità di bit multipla di quella di `char`
- Questa relazione rispecchia esattamente il fatto che ciascuno dei formati standard di un *ASM-PM* definisce parole che formano aree di memoria di una determinata dimensione

Tipi di Dato Aritmetici

- Ciascuna implementazione di C Standard, dunque, associa ad ogni tipo di dato aritmetico uno dei propri formati di dato
- Esempio tipico
  - `short int` associato ad un formato di dimensione 2
  - `long int` associato ad un formato di dimensione 4
  - `long long int` associato ad un formato di dimensione 8
  - `float` associato al formato per dati floating point con precisione minore
  - `double` e `long double` associati a formati per dati floating point di precisione crescente, se esistono in M
  - `int` associato al formato più efficiente, o preferibile per altro motivo, tra tutti quelli ai quali può applicarsi un'interpretazione di dato per interi

Tipi Aritmetici

- Code1_m68k mostra una traduzione MC68000 di Code1_pc, in cui il tipo `unsigned` è associato al formato `word`

Code1_pc

```
unsigned a = 7, b = 2;
if ( 3 * b > a ) a += 4;
goto elsewhere;
```

Code1_m68k

```
    move.w    b,d0
    mulu   #3,d0      ; unsigned
    cmp.w  a,d0
    bls    no_add     ; unsigned
    add.w  #4,a
no_add:
    jmp    elsewhere
    org    $4000
a:  dc.w   7
b:  dc.w   2
```

Tipi Aritmetici

- Code2_m68k mostra una traduzione MC68000 di Code2_pc, in cui il tipo `int` è associato al formato `word`

Code2_pc

```
int a = 7, b = 2;
if ( 3 * b > a ) a += 4;
goto elsewhere;
```

Code2_m68k

```
    move.w    b,d0
    muls   #3,d0      ; signed
    cmp.w  a,d0
    ble    no_add     ; signed
    add.w  #4,a
no_add:
    jmp    elsewhere
    org    $4000
a:  dc.w   7
b:  dc.w   2
```

Tipi Aggregati

- Una variabile che ha un tipo aggregato è, sostanzialmente, un gruppo di variabili alle quali si può accedere individualmente
- Si potrebbe ipotizzare di memorizzare una variabile di un tipo aggregato mediante un insieme di parole non correlate tra di loro
- Ad esempio, una variabile di un tipo aggregato potrebbe essere memorizzata in parole di memoria che abbiano indirizzi totalmente indipendenti gli uni dagli altri

Tipi Aggregati

- Code3_pc è un frammento di programma C che definisce e usa una variabile di tipo array; Code3_m68k_1 ne è una traduzione che memorizza l'array in `word` non correlate tra loro

Code3_pc

```
int x[ 4 ] = { 7, 0, 3, 8 };
x[ 1 ] = x[ 3 ] + x[ 0 ];
goto elsewhere;
```

Code3_m68k_1

```
    move.w   x3,d0
    add.w    x0,d0
    move.w   d0,x1
    jmp elsewhere
    org      $4000
x2: dc.w     3, 2
x0: dc.w     7
x3: dc.w     8
x1: dc.w     0
```

Tipi Aggregati

- Questa semplice idea presenta seri problemi, soprattutto nella rappresentazione di array
  1. Le istruzioni *ASM* accedono a parole di memoria tramite il modo di indirizzamento diretto-memoria, e quindi indicano in modo esplicito e statico gli indirizzi delle parole; di conseguenza non è possibile tradurre espressioni C in cui si accede ad elementi di un array con indici calcolati a tempo di esecuzione
  2. Nel programma *ASM* è necessario specificare molti indirizzi non correlati tra loro, uno per ciascun elemento dell'array
- Il più importante dei due problemi è il primo, la cui soluzione richiede due nuove idee

Tipi Aggregati

- La prima idea è *memorizzare una variabile di tipo aggregato in un'area di memoria*
  - Tutti gli elementi di una variabile di un tipo aggregato, vengono memorizzati all'interno della stessa area di memoria
  - Un array viene memorizzato in un'area di memoria, all'interno della quale
    - le parole che memorizzano i singoli elementi, sono ordinate in base agli indici degli elementi stessi
    - non vi sono byte inutilizzati o utilizzati per memorizzare dati diversi dagli elementi dell'array
- Questa idea è utile anche da sola, in quanto permette di risolvere il secondo dei problemi evidenziati in precedenza

Tipi Aggregati

- Usando l'idea precedente nel memorizzare un array, si stabilisce una relazione tra gli indirizzi delle parole che memorizzano ciascuno degli elementi dell'array
- Se ciascun elemento dell'array è memorizzato in un byte e se l'elemento di indice 0 è memorizzato nel byte che ha indirizzo A, si ha che per ciascun valore positivo k, l'elemento di indice k (se esiste) è memorizzato nel byte che ha indirizzo pari a

$$A + k$$

- Se ciascun elemento dell'array è memorizzato in una parola di dimensione m e se l'elemento di indice 0 è memorizzato nella parola che ha indirizzo A, si ha che per ciascun valore positivo k, l'elemento di indice k (se esiste) è memorizzato nella parola che ha indirizzo pari a

$$A + k \cdot m$$

Tipi Aggregati

- Grazie a tale relazione, si possono esprimere tutti gli indirizzi delle parole che memorizzano gli elementi dell'array, in funzione dell'indirizzo della parola che memorizza l'elemento di indice 0, migliorando la leggibilità del programma *ASM*
- Alcuni *ASM*, consentono di utilizzare tale relazione in espressioni costruite applicando degli operatori aritmetici a label e rappresentazioni numeriche
- In fase di traduzione, le label vengono sostituite dai valori numerici degli indirizzi, viene valutata l'espressione numerica, e il risultato viene usato come indirizzo
- Attraverso queste espressioni si esprimono in modo piuttosto leggibile anche gli indirizzi di parole non sono identificate da label, permettendo di diminuire la quantità di label nel codice sorgente

Tipi Aggregati

- Code3_m68k_2 è una traduzione di Code3_pc, che memorizza l'array usando la precedente idea ed esprime gli indirizzi delle `word` che memorizzano gli elementi dell'array, in funzione dell'indirizzo della `word` che memorizza l'elemento di indice 0

Code3_pc

```
int x[ 4 ] = { 7, 0, 3, 8 };
x[ 1 ] = x[ 3 ] + x[ 0 ];
goto elsewhere;
```

Code3_m68k_2

```
     move.w   x+6,d0
     add.w    x,d0
     move.w   d0,x+2
     jmp elsewhere
     org      $4000
x:   dc.w     7,0,3,8
```

Tipi Aggregati

- È opportuno ribadire che le espressioni aritmetiche con label sono statiche, ovvero vengono calcolate dall'assembler a tempo di traduzione
- Nel codice sorgente, gli operatori aritmetici che formano tali espressioni vengono indicati con i tipici simboli degli operatori aritmetici (+, -, *, /) per evidenziare che non si tratta di istruzioni dell'*ASM*, ma di comandi per l'assemblatore
- Di conseguenza, tali espressioni possono contenere solo label e valori numerici costanti
- È pertanto errato tentare di usarle per tradurre accessi ad elementi di un array che hanno indice dinamico, ovvero costituito da un'espressione contenente variabili

Tipi Aggregati

- Code4_pc è un frammento di programma C in cui si accede ad un elemento di un array con un indice costituito da una espressione contenente una variabile; Code4_m68k_1 è un tentativo errato di traduzione che utilizza un'espressione aritmetica statica per tentare di accedere all'elemento che ha indice dinamico

Code4_pc

```
int x[ 4 ] = { 7, 0, 3, 8 };
unsigned k;
// k calcolato a run-time
x[ 1 ] = x[ k ] + x[ 0 ];
goto elsewhere;
```

Code4_m68k_1

```
* k calcolato a run-time
    move.w    x+k+k,d0 * errore
    add.w     x,d0
    move.w    d0,x+2
    jmp    elsewhere
    org       $4000
x:  dc.w      7,0,3,8
k:  dc.w      0
```

Tipi Aggregati

- Per tradurre espressioni C in cui compaiono elementi di un array con indice specificato da un'espressione calcolata a tempo di esecuzione, è necessario calcolare dinamicamente gli indirizzi delle parole che memorizzano tali elementi
- Tale calcolo va fatto mediante la relazione tra gli indirizzi delle parole che memorizzano gli elementi di un array, come negli esempi precedenti
- La differenza è che il calcolo deve essere fatto dinamicamente invece che staticamente, ovvero mediante istruzioni *ASM*
- L'indirizzo calcolato, viene memorizzato in una parola di *M*

Tipi Aggregati

- Qui entra in gioco una seconda idea fondamentale: *memorizzare in una parola P, l'indirizzo di un'altra parola di memoria che contiene un dato cui si è interessati*, ovvero *usare gli indirizzi come dati*
- La parola P viene chiamata *puntatore* a un dato
- Attraverso un puntatore P si accede alla memoria attraverso un indirizzo che non è statico, ma che, essendo il contenuto di P, può essere calcolato e modificato dinamicamente
- Tale tipo di accesso viene chiamato *indiretto* ed è una tecnica di programmazione di importanza fondamentale

Tipi Aggregati

- Se D è una parola di memoria che contiene un dato e P è una parola che ha un formato a cui può essere associata un'interpretazione di dato per indirizzi, è possibile fare in modo che P diventi un puntatore a D
- L'operazione che memorizza in P l'indirizzo di D, si chiama *address-of* o *referenziazione*: l'effetto dell'operazione è che P contiene un *riferimento* a D
- L'operazione che permette di accedere a D attraverso P, si chiama *indirezione* o *dereferenziazione*: l'effetto è accedere a D (leggendone o modificandone il contenuto) senza specificarne direttamente l'indirizzo, ma sfruttando il fatto che tale indirizzo è memorizzato in P

Tipi Aggregati

- Tutti i linguaggi *ASM*, permettono di effettuare un accesso indiretto ad una parola di memoria, attraverso uno o più specifici modi di indirizzamento
- Ad esempio, sia MC68000 che MIPS32 permettono di utilizzare un registro come puntatore, e dispongono di un modo di indirizzamento chiamato *indiretto-registro*, che consente di eseguire un'operazione di indirezione
- Nel caso di MC68000 il registro che contiene l'indirizzo dell'operando, deve essere un registro indirizzi
- In entrambi gli *ASM*, lo specificatore di operando per il modo di indirizzamento indiretto-registro, è formato dal nome del registro racchiuso tra parentesi tonde

Tipi Aggregati

- Code4_m68k_2 è una traduzione corretta di Code4_pc, che calcola dinamicamente l'indirizzo della `word` che memorizza l'elemento dell'array di indice `k` e utilizza il modo di indirizzamento indiretto-registro per accedere a tale elemento

Code4_m68k_2

```
* k calcolato a run-time
    move.l    #x,a0      ; a0 è un riferimento a x[0]
    move.w    k,d1       ; valore di k
    mulu      #2,d1      ; moltiplica k per 2
    add.w     d1,a0      ; a0 è un riferimento a x[k]
    move.w    (a0),d0    ; indirezione su a0: accesso a x[k]
    add.w     x,d0       ; somma x[0] a x[k]
    move.w    d0,x+2
    jmp    elsewhere
    org       $4000
x:  dc.w      7,0,3,8
k:  dc.w      0
```

Tipi Aggregati

- C Standard obbliga le implementazioni a memorizzare ciascun array in un'area di memoria
  - per consentire di accedere in modo efficiente ad elementi di un array con indice specificato da un'espressione calcolata a tempo di esecuzione
  - perché in generale la gestione di variabili che hanno tipo aggregato è più efficiente se esse sono memorizzate in aree di memoria
- In particolare, C Standard prescrive che gli elementi di un array vengano memorizzati ciascuno in una propria area, contenuta in un'unica area per l'intero array, senza che vi siano *byte* inutilizzati tra quelli che memorizzano un elemento e quelli che memorizzano il successivo

Tipi Aggregati

- C Standard stabilisce inoltre che ogni variabile di un tipo struttura sia memorizzata in un'area di memoria
- All'interno di tale area, i membri della struttura vengono memorizzati nell'ordine in cui sono dichiarati nel codice sorgente
- Tuttavia è possibile che tra un membro e l'altro vi siano dei *byte* inutilizzati, per permettere un allineamento ottimale dei membri in memoria
- Per lo stesso motivo è possibile che vi siano *byte* inutilizzati al termine dell'area

Tipi Aggregati

- Ad esempio consideriamo una struttura formata da un membro di tipo `char` seguito da uno di tipo `long`
- Una implementazione MC68000 potrebbe utilizzare un *byte* per il primo membro e una *long* per il secondo
- Il formato *long* ha il vincolo di avere indirizzi multipli di 2 (ovvero *pari*)
- La soluzione più semplice ed efficiente dal punto di vista della velocità di esecuzione è
  - Memorizzare la struttura in un'area di memoria di 6 byte che ha indirizzo pari
  - Usare il primo *byte* dell'area per memorizzare il primo membro
  - Non utilizzare il secondo byte
  - Usare i restanti 4 *byte* per il secondo membro

Tipi Aggregati

- Per garantire la massima flessibilità in applicazioni particolari (in genere a prezzo di una minore velocità di esecuzione), C Standard consente anche di definire strutture i cui membri hanno quantità di bit diverse da quelle utilizzate per i tipi base, che quindi potrebbero non essere associati a uno dei formati standard di un *ASM-PM*
- Tali membri sono chiamati *bit-field* e sono analoghi ai bit-field disponibili in alcuni *ASM-PM*
- I bit-field nei tipi struttura, verranno approfonditi in una futura presentazione

Sessione di Esercizi: Organizzazione dei Dati in Memoria

- Svolgere il gruppo di esercizi **Organizzazione dei Dati in Memoria** su *Edu99*
- Per ulteriori spiegazioni sui contenuti degli esercizi, si vedano
  - esempi *Assembly Pointers* e *Assembly Arrays 1* su *Edu99*
  - **[M68000]**
  - **[MIPS32]**