Fondamenti di Informatica
Ingegneria Clinica

17 Dicembre 2009

Raffaele Nicolussi

FUB - Fondazione Ugo Bordoni

Via B. Castiglione 59 - 00142 Roma

Docente	**Raffaele Nicolussi**	*rnicolussi@fub.it* *0654803323*
Lezioni **Aula 54 (ex aula 4)** **Via del Castro Laurenziano, 7**	**Lunedì, Giovedì, Venerdì**	**12:00 – 13:30**
Esercitazioni **Aula 15** **Via Tiburtina, 205**	**Giovedì**	**14:00 – 16.30**
Ricevimento:	**Per appuntamento**	**in FUB, per email, per telefono**
Sito web:	**http://w3.uniroma1.it/IngClinFondinf**	

Le strutture

Rappresentazione di dati disomogenei

Rappresentazione di dati eterogenei

Ipotesi: definizione di insieme di oggetti di tipo diverso (es. int, float, array)

Esempio:

- **Rappresentazione di un individuo**
  - **nome: array di caratteri**
  - **cognome: array di caratteri**
  - **data di nascita: tre interi**
  - **codice fiscale: array di 17 caratteri**

→ queste informazioni non possono essere inserite in un array perché i tipi sono eterogenei

→ memorizzazione in variabili distinte

```
char nome[15], cognome[15], cod_fis[17];
short int giorno, mese, anno;
```

Rappresentazione di dati eterogenei

L'immissione dei dati dovrebbe avvenire indipendentemente per ciascun campo:

```
strcpy (nome, "Mario");
strcpy (cognome, "Rossi");
strcpy(cod_fis, "RSSMRO63L33H501S");
giorno = 12;
mese = 7;
anno = 1963;
```

→ **organizzazione di dati piuttosto disomogenea: le informazioni relative ad una persona sono sparse per la memoria**

→**gestire un database sarebbe alquanto complicato …:**

```
char nome[1000][15], cognome[1000][15], cod_fis[1000][17];
short int giorno[1000], mese[1000], anno[1000];
```

Strutture

Soluzione: un dato in grado di contenere informazioni di tipo diverso.

→ In C esiste un tipo di dato, chiamato struttura, che permette di definire in modo aggregato tipi di dato eterogenei.

- **Una struttura consiste di un nome e di componenti (campi o elementi).**
  - **Il nome assegnato alla struttura diventa il nome del tipo di dato**
    - **tipo di dato definito dall'utente che può essere usato per dichiarare variabili di quel tipo.**

- **Nota Bene: la struttura da' la possibilità all'utente di definire un tipo di dato e di denominarlo**

Strutture: sintassi

```
struct <tag_name>
{
    <tipo1> <nome_campo1>;
    <tipo2> <nome_campo2>;
    ……..;
    <tipoN> <nome_campoN>;  };
```

etichetta

campi o elementi

Esempio:

```
struct contribuente {
        char nome[15];
        char cognome[15];
        char cod_fis[17];
        short int giorno;
        short int mese;
        short int anno;
        };
```

Strutture

- Una struttura è nuovo tipo di dato che si può usare per definire variabili usando l'etichetta struct contribuente esattamente come int, float , etc.

- Definizione di una variabile cont1 di tipo struct contribuente

```
struct contribuente cont1;
```

```
struct contribuente {
    char nome[15], cognome[15], cod_fis[17];
    short int giorno, mese, anno;
            } cont1;
```

Strutture

cont1 è di tipo struct contribuente

- **è formata dagli stessi campi della struttura**
- **in associazione al tipo vengono fornite dal C delle funzioni di accesso ai singoli campi**
    - **per fornire valori ai campi**
    - **per prelevare i valori dei campi**
- **il meccanismo di memorizzazione di una variabile di tipi struttura tiene conto dell'unicità della struttura**

- **Si noti la differenza con l'array:**
  - **quando definiamo un array, l'istanza è unica**
  - **Una struttura può avere più di un istanza**

struct contribuente mrossi;

struct contribuente pallini;

struct contribuente torto;

nome	cognome	cod_fis	giorno	mese	anno

struct contribuente mrossi;

Mario	Rossi	RSSMRO52P34H501Y	18	8	1952

struct contribuente pallini;

Carlo	Pallini	PLNCRL60F80H501W	8	8	1960

struct contribuente torto;

Gelso	Torto	TRTGLS76J08H501W	6	6	1976

Meccanismo di accesso ai campi

<nome_struttura>.<nome_campo>

mrossi.nome

pallini.giorno

torto.cod_fis

mrossi.anno

Assegnazione valori

mrossi.nome <- “Mario Rossi”

pallini.giorno <- 8

torto.cod_fis <- “TRTFRC76J08H501W”

mrossi.anno <- 1960

Memorizzazione

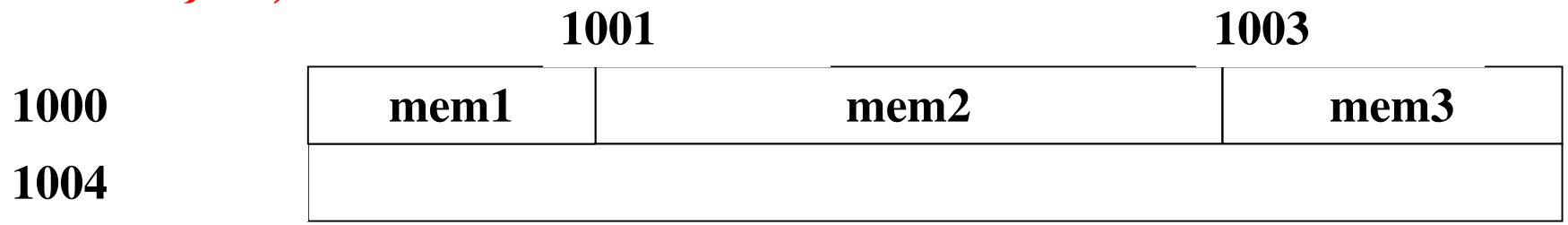
- **Dichiarando una struttura, ci sarà un'allocazione di memoria pari alla somma di quella richiesta per i campi.**
- **La memorizzazione della struttura avviene in modo consecutivo nell'ordine in cui i campi sono dichiarati.**

```
struct contribuente   {
    char nome[15], cod_fis[17];
   short int giorno, mese, anno;   };
struct contribuente cont1;
```

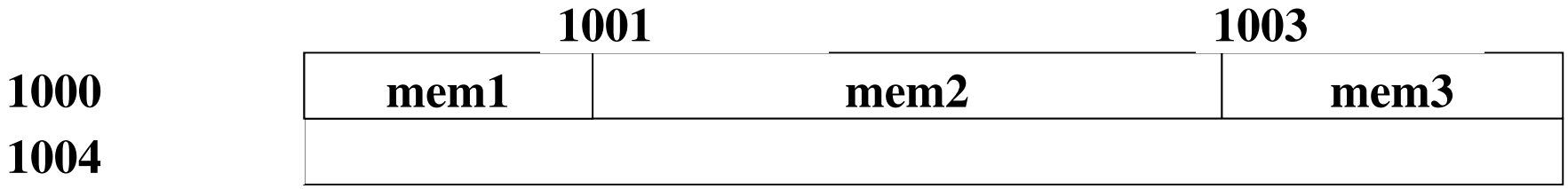
Memorizzazione

- La contiguità degli elementi della struttura non è assicurata per la presenza di spazi vuoti tra gli elementi a causa dell'allineamento degli elementi
  - Es: tutti gli oggetti di dimensioni superiori a char sono memorizzati in locazioni con indirizzi pari.
- Queste restrizioni sull'allineamento possono creare spazi vuoti (buchi o gap) nelle strutture.

```
struct esempio_allineamento {
        char mem1;
        short mem2;
        char mem3;
        } s1;
```



- In mancanza di restrizioni di allineamento, **s1 viene memorizzato senza buchi.**
- In presenza di restrizioni, vi sarà un **buco tra mem1 e mem2.**



- **Confrontare due strutture è un errore di sintassi.**

Composizione di una struttura

- I membri delle strutture possono contenere variabili di qualsiasi tipo scalare (int, float, etc.) o aggregato come array e altre strutture.
  - Una struttura non può contenere una istanza di se stessa.
- Una struttura può contenere un elemento che sia un puntatore (anche a se stessa)

- A partire da una struttura ST possiamo dichiarare
  - variabili di tipo ST (istanze)

    ```
    struct ST istance1;
    ```
  - puntatori alla struttura

    ```
    struct ST *s;
    ```
  - vettori di elementi di tipo struttura

    ```
    struct ST vettore[1000];
    ```

struct contribuente contrs[1000], *pctr;

- **contrs[1000]:**
  - **array di 1000 elementi**
  - **ogni elemento è una struttura con i campi nome, cognome, cod_fis, giorno, mese, anno.**

- **pctr: un puntatore ad un struttura**
  - **pctr= &contrs[10] fa puntare pctr all'undicesimo elemento dell'array.**

contrs[0]

- contrs [2]

nome	cognome	cod_fis	giorno	mese	anno
nome	cognome	cod_fis	giorno	mese	anno
nome	cognome	cod_fis	giorno	mese	anno
nome	cognome	cod_fis	giorno	mese	anno

ptr →

nome	cognome	cod_fis	giorno	mese	anno

nome	cognome	cod_fis	giorno	mese	anno
nome	cognome	cod_fis	giorno	mese	anno
nome	cognome	cod_fis	giorno	mese	anno

```
contrs[997].nome =”Maria”;
contrs[997].cod_fis =”BNCMRA78F89H505Y”;
contrs[997].giorno =15;
contrs[997].mese=9;
contrs[997].anno =1964;
```

Accesso diretto (1)

- **nome istanza della struttura e il nome del membro separati dall'operatore punto (.), detto anche operatore membro di struttura**

  cnt1.nome, cnt1.giorno, cnt1.mese …

- **Un costrutto della forma**

  variabile_di_struttura.nome_membro

- **è una variabile e può essere utilizzata nello stesso modo di una variabile semplice o un elemento di un array.**

```
if (cnt1.giorno > 31 || cnt1.mese >12 )
    printf("data errata\n");
```

- **Può anche essere utilizzata per assegnare valori ai singoli membri.**

```
struct contribuente cnt3;
strcpy(cnt3.nome,"Luigi");
cnt3.giorno = 23;
```

Accesso diretto (2)

- **È possibile assegnare una intera struttura ad un'altra**

```
struct contribuente cnt4;
cnt4=cnt3;
```

- **È equivalente all'assegnazione campo per campo.**

Accesso indiretto tramite puntatore (1)

- La definizione di puntatore ad una struttura è uguale alle altre dichiarazioni di puntatori:

  struct contribuente *Pcnt;

- Per accedere ad un elemento si usa l'**operatore freccia**, o **operatore puntatore a struttura**
  - **operatore freccia ->**
  - simbolo **meno** (-) seguito senza spazi dal simbolo di **maggiore** (>)
- **Accesso:**
  - nome del puntatore seguito dall'operatore freccia e dal nome del campo

Pcnt->nome, Pcnt->giorno, Pcnt->cod_fis, …

Accesso indiretto tramite puntatore (2)

- **Può essere usato esattamente come l'altra forma di accesso**

```
if (Pcnt->giorno > 31 || Pcnt->mese >12 )
    printf("data errata\n");
```

- **L'operatore freccia è una forma abbreviata per accedere all'indirizzo contenuto nel puntatore e quindi applicare poi l'operatore punto:**

  Pnct -> equivale a ***Pcnt**

  Pcnt->nome equivale a **(*Pcnt).nome**

- **Si noti che le parentesi tonde sono necessarie perché l'operatore membro di struttura (il punto) ha priorità maggiore dell'operatore di risoluzione del riferimento (l'asterisco).**

Esempio 10.2 Deitel&Deitel

```
#include <stdio.h>
struct carta {
  char *numero, *seme;  };
int main (void)   {
  struct carta a, *aPtr;
  a.numero=”Asso”;
  a.seme=”Picche”;
 aPtr=&a;
 printf("Stampa con operatore membro di struttura\n");
 printf(“%s %s %s\n”, a.numero, “di”, a.seme);
 printf("Stampa con operatore puntatore a struttura\n");
 printf(“%s %s %s\n”, aPtr->numero, “di”, aPtr->seme);
 printf("Stampa con punt. e operatore membro di struttura\n");
 printf(“%s %s %s\n”, (*aPtr).numero, “di”,(*aPtr).seme);
 return 0; }
```

10.2.c

Modi alternativi per definire una struttura

- Una struttura può essere dichiarata anche **senza nome**:

```
struct  {
    char nome[15], cognome[15], cod_fis[17];
    short int giorno, mese, anno;
        } cont1;
```

- Nessuna istanza di questa struttura può essere dichiarata al di fuori della lista di variabili ammesse dopo la parentesi graffa:

```
struct {
    char nome[15], cognome[15], cod_fis[17];
    short int giorno, mese, anno;
    } cont1, contrs[1000], *pctr;
```

Uso del typedef

Rappresenta un modo per creare dei sinonimi per i tipi definiti in precedenza

typedef struct contribuente Contribuente;

Contribuente è il sinonimo

Rappresenta l'intera struttura, compresa la parola chiave struct

Uso del typedef

```
struct contribuente {
    char nome[15], cognome[15], cod_fis[17];
    short int giorno, mese, anno;  };
typedef struct contribuente Contribuente;
```

```
typedef struct  [contribuente]
{
char nome[15], cognome[15], cod_fis[17];
short int giorno, mese, anno;
}  Contribuente;
```

Uso del typedef

- **Per dichiarare una variabile del tipo struttura si usa solo il nome assegnato con il typedef:**

  Contribuente c1;

  equivalente a

  struct contribuente c1;

- **typedef crea solo un sinonimo per un tipo già dichiarato.**

- **typedef può essere usato per creare sinonimi di tipi fondamentali per garantire la portabilità di un programma.**

Uso del typedef

- **Esempio: programma scritto per funzionare con interi di 4 byte potrebbe usare int su un sistema e long su un altro, a seconda dell'implementazione.**

  typedef Integer int;

  typedef Integer long;

- **Tutto il programma può essere scritto usando**

  Integer

- **Si deve cambiare solo il typedef a seconda se quel sistema abbia o no gli int di 4 byte**

Inizializzazione

- Analoga a quanto avviene per gli array
  - nome della variabile del tipo struttura + simbolo uguale "=" + la lista dei valori tra parentesi graffe
  - Ogni valore iniziale deve essere dello stesso tipo del corrispondente tipo della struttura.

```
Contribuente cnt1 = { "Mario Rossi", "RSSMRO63L33H501S",
                      12, 7, 1963};
struct contribuente cnt1 = { "MarioRossi",
                      "RSSMRO63L33H501S",  12, 7, 1963};
```

Inizializzazione

- **Elenco con meno valori di quanti siano i membri della struttura:**
  - **i rimanenti sarebbero inizializzati automaticamente a 0 (NULL nel caso di puntatori)**
- **Elenco con più valori di quanti siano i membri della struttura:**
  - **errore di sintassi**

E’ possibile inizializzare contestualmente alla dichiarazione

```
struct contribuente {
   char nome[15], cognome [15], cod_fis[17];
   short int giorno, mese, anno;
    } cnt1 = { “Mario”,  “Rossi”,
                    “RSSMRO63L33H501S”, 12,
                    7, 1963};
```

- **Una struttura può anche dichiarata al di fuori di un blocco funzione ed essere quindi globale per l'intero programma**

```
typedef  struct risultato {
                int uno;
                int due; } Risultato;
typedef struct partita {
        char squadra_casa[20], squadra_ospite[20], arbitro[20];
        Risultato ris;} Partita;
```

- **Le variabili di struttura dichiarate nel programma al di fuori di una funzione sono automaticamente inizializzate a 0 (NULL nel caso di puntatori), se non sono state altrimenti inizializzate.**

- **Le variabili di struttura possono anche essere inizializzate con istruzioni di assegnamento.**

```
Partita derby;
derby.squadra_casa = “Lazio”;
derby.squadra_ospite = “Roma”;
derby.arbitro = “Non lo so”;
derby.ris.uno = 1;
derby.ris.due = 5;
```

Nome dei campi

- I nomi dei campi all'interno di una struttura devono essere unici, ma campi di strutture differenti possono avere lo stesso nome.
  - questo non crea ambiguità perché i metodi di accesso ad una struttura usano sempre il nome della struttura,

```
struct frutta {
    char *nome;
    int calorie;  };

struct vegetale  {
    char *nome;
    int calorie; };

struct frutta a;                a.nome
struct vegetale c, d;           c.nome  c.calorie
```

- **Dati**
  - **Gruppo di contribuenti memorizzati con nome, codice fiscale, data di nascita (giorno, mese, anno)**
  - **1000 contribuenti memorizzati in un array**
- **Problema**
  - **Contare il numero di persone che hanno un'età compresa tra due valori dati**
  - **Supponiamo di avere definito una funzione leggi per memorizzare l'array**

Definizione di contribuente

```
struct contribuente {
        char nome[15], cod_fis[17];
        short int giorno, mese, anno;     };
typedef struct contribuente Contribuente;
```

Definizione matrice dei contribuenti

```
Contribuente gruppo [DIM];
```

```
#include <stdio.h>
#define DIM 1000
int conta_persone (Contribuente [ ], int, int, int);
int leggi (Contribuente [ ]);
int main (void)                              {
    struct contribuente {
            char nome[15], cod_fis[17];
            short int giorno, mese, anno;     };
    typedef struct contribuente Contribuente;
    Contribuente gruppo [DIM];
    int min=20, max=45, a_cor=2002,  num_persone=0;
    leggi(gruppo);
    num_persone =conta_persone (gruppo, min, max ,a_cor);
    printf("Persone di eta' compresa tra %d e %d anni: %d\n", min, max,
    num_persone);
   return 0;}
```

```
int conta_persone (Contribuente gruppo [], int low_age,
                   int high_age, int current_year)
{
  int i, age, count = 0;

  for (i = 0; i < size; ++i)
  {
    age = current_year - gruppo[i].anno;
    if (age >= low_age && age <= high_age) count++;}
  return count;
}
```

Utilizzo di strutture con funzioni

- **Le strutture possono essere passate come parametri a funzioni e possono a loro volta essere restituite dalle funzioni stesse.**
  - **intera struttura**
  - **singoli membri**
  - **puntatori a una struttura**
- **Quando viene passata una struttura o i suoi membri, il passaggio è per valore: la struttura o i suoi membri non sono modificati**
  - **copia locale nel corpo della funzione chiamata.**
  - **se uno dei membri della struttura è un array, anch'esso viene copiato.**
    - **il passaggio di una struttura per valore può risultare inefficiente.**
- **Alternativa: passaggio della struttura per riferimento con lo indirizzo della variabile di struttura.**

Esempio:

```
typedef struct {
    char            nome [25];
    int             mat_impiegato;
    struct dept     sezione;
    double          salario;
    … … ……
} impiegato;
```

- Si deve definire struct dept prima dell'uso perché il compilatore deve conoscere la dimensione di ogni membro della struttura

```
struct dept {
    char            nome_sezione;
    int             numero_sezione;
};
```

- Funzione che aggiorni le informazioni sugli impiegati.

```
void aggiorna (impiegato *p)
{    ……..……
     printf (“Inserisci il numero della sezione:   \n”);
     scanf (“%d”, &n);
     p -> sezione.numero_sezione = n;
  …….. }
```

- p -> sezione.numero_sezione è equivalente a
  (p -> sezione).numero_sezione
- si accede, tramite un puntatore, a un campo
- aggiorna ( ) può essere usata chiamata così:

```
impiegato e;
aggiorna(&e);
```

- viene passato l’indirizzo di e
- Non è quindi necessario effettuare alcuna copia locale della struttura in aggiorna ( )

- **Altrimenti modifica all'interno della funzione (o main) che contiene la struttura**
- **Accesso a un membro di una struttura all'interno di una struttura:**

  e.sezione.numero_sezione

- **che è equivalente a (e.sezione).numero_sezione**

Strutture nidificate

- Una struttura nidificata è una struttura un cui almeno uno degli elementi è a sua volta una struttura.
- La precedente struttura contribuente potrebbe essere definita così:

```
typedef struct {
char nome[15], cod_fis[17];
struct  {
        short int giorno,  mese, anno;  } data_nascita;
} Contribuente;
```

- I tre campi che rappresentavano la data di nascita sono sostituiti da una struttura che contiene tre elementi.
- L'allocazione di memoria rimane inalterata, ma invece di accedere all'anno come cont1.anno si deve scrivere

  cont1.data_nascita.anno