

Struct, enum, Puntatori e Array dinamici

Tratti dal corso del
Dr. Francesco Fabozzi
Corso di Informatica

Tipi di dato in C

- Abbiamo esaminato diversi tipi di dato che il C mette a disposizione dell'utente
 - Tipi elementari
 - int, char, float, ...
 - Tipi complessi
 - Vettori, puntatori
- Il C permette all'utente di definire nuovi tipi di dato

typedef

- Il modo più semplice per definire un nuovo tipo in C è tramite la parola chiave

typedef

- `typedef tipo nuovo_nome_tipo`
- Un tipo di dato esistente potrà essere riferito sia con `tipo` sia con `nuovo_nome_tipo`
- Usato per rendere il codice più leggibile

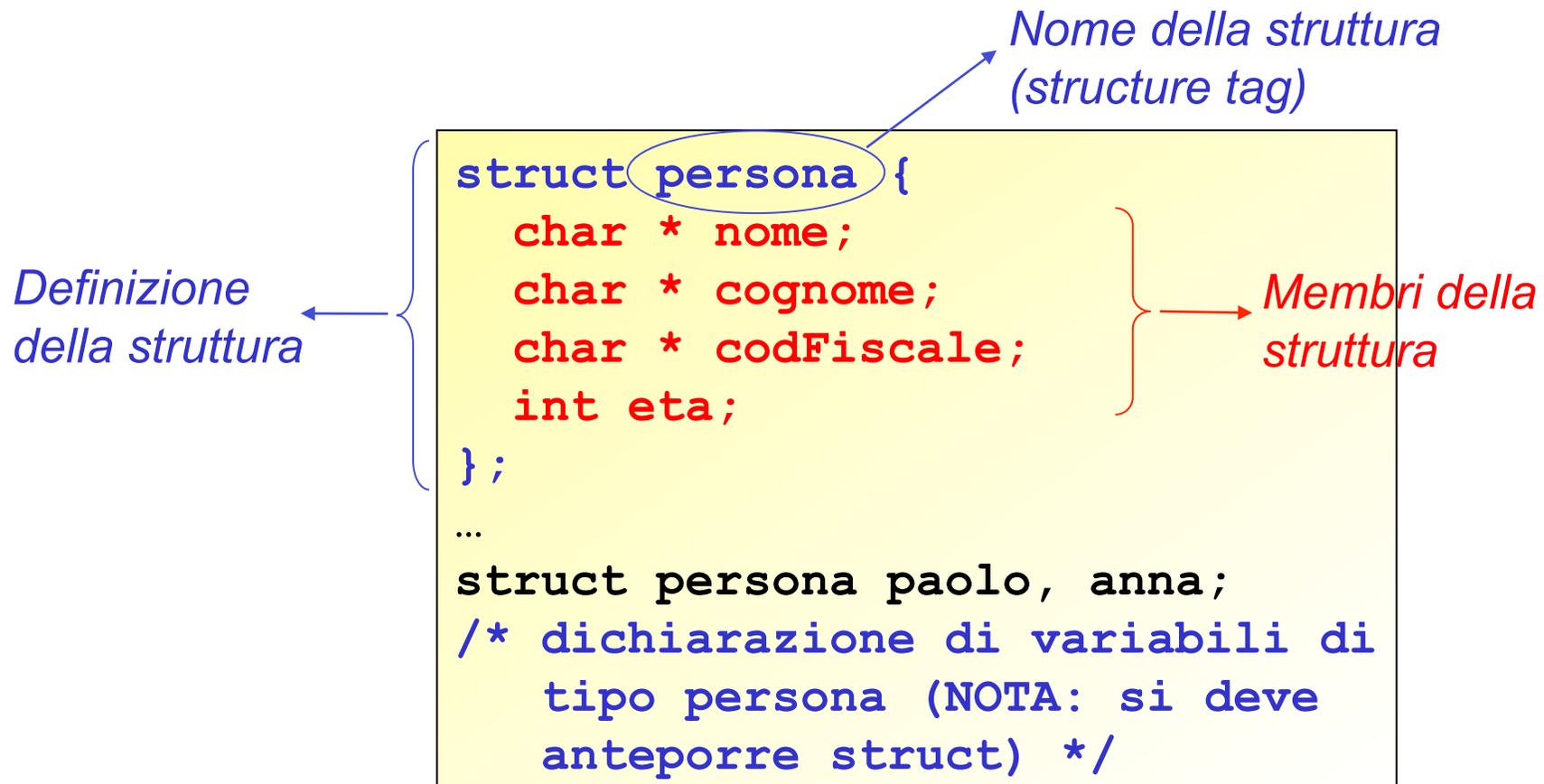
```
typedef float coordinata
coordinata x = 2.1;
coordinata y = 1.0;
/* x e y sono due float che saranno usati per
   rappresentare le coordinate di punti */
```

Strutture

- Un gruppo di variabili logicamente correlate possono essere incapsulate in un tipo di dato più complesso detto *struttura*
 - Le strutture costituiscono dei tipi derivati perchè sono ottenute utilizzando altri tipi
- La definizione di una struttura è introdotta dalla parole chiave **struct**

Strutture: definizione

- Esempio di struttura: persona



Strutture: definizione

- Le variabili di tipo struttura possono essere dichiarate insieme alla definizione della struttura
- Lo structure tag è opzionale
 - Ma è buona abitudine metterlo
 - Se manca lo structure tag non è possibile dichiarare variabili struttura nel resto del programma

```
struct {  
    const char * nome;  
    const char * cognome;  
    const char * codFiscale;  
    int eta;  
} paolo, anna;
```

Strutture: dichiarazione

- Le variabili di tipo struttura sono dichiarate come qualsiasi variabile antepo-
nendo però la keyword **struct**
 - Anche puntatore a una struttura e vettore di strutture

```
struct persona paolo, anna;  
/* dichiarazione di variabili di  
   tipo persona */  
  
struct persona * cliente;  
/* puntatore a una persona */  
  
struct persona studenti[10];  
/* array di persone */
```

Strutture: dichiarazione

- Per snellire le dichiarazioni (evitando di usare la keyword **struct**) si preferisce definire un nuovo tipo dalla struttura
 - Si fa un typedef prima delle dichiarazioni

```
struct persona {  
...  
};  
typedef struct persona Persona;  
/* definisce il nuovo tipo Persona */  
  
Persona paolo, anna;  
Persona studenti[10];
```

Strutture: dichiarazione

- Ancora più compatto: definizione della struttura e typedef uniti insieme

```
typedef struct {  
...  
} Persona;  
/* definisce il nuovo tipo Persona */  
  
Persona paolo, anna;  
Persona studenti[10];
```

Strutture: membri

- L'accesso ai membri di una struttura è effettuato tramite l'operatore `.` oppure `->`
 - Si usa `.` nel caso di una struttura
 - Si usa `->` nel caso di un puntatore a una struttura

```
scanf( "%s", anna.nome )
/* il nome di anna viene inizializzato
   dallo standard input */
cliente->eta++;
/* si aumenta di 1 l'età del cliente */
printf( "%s", studente[5].cognome );
/* stampa il nome di uno studente
   sullo standard output */
```

Strutture membri di strutture

- E' possibile nidificare le strutture

```
typedef struct {
    int giorno;
    int mese;
    int anno;
} Data;

typedef struct {
    const char * nome;
    const char * cognome;
    Data dataNascita;
} Persona;

...
Persona squadra[20];
printf ("%d"; squadra[5].dataNascita.anno);
```

Esempio: area di cerchi

```
/* programma che stampa l'area di cerchi
   e la distanza del loro centro dall'origine */
#include <stdio.h>

typedef struct{
    float x;
    float y;
} Punto;

typedef struct{
    Punto centro; /* centro del cerchio */
    float raggio;
} Cerchio;

float areaCerchio( float r );
/* funzione che calcola l'area di un cerchio */
float distanza( Punto p1, Punto p2 );
/* funzione che calcola la distanza tra due punti */
/* continua... */
```

Esempio: area di cerchi

```
main() {
    Cerchio c;
    Punto orig;
    orig.x = 0; orig.y = 0;
    printf("Inserisci 0 per terminare \n");
    printf("Inserisci il raggio \n");
    scanf("%f", & c.raggio);
    while( c.raggio != 0 ) {
        printf("Inserisci la coordinata x del centro \n");
        scanf("%f", & c.centro.x);
        printf("Inserisci la coordinata y del centro \n");
        scanf("%f", & c.centro.y);
        printf("area cerchio = %f \n",
            areaCerchio( c.raggio ));
        printf("distanza dall'origine = %f \n",
            distanza( c.centro, orig ));
        printf("Inserisci il raggio \n");
        scanf("%f", &c.raggio);
    }
}
```

Esempio: area di cerchi

```
/* funzione che calcola la distanza tra
   due punti del piano */

#include <math.h>

float distanza( Punto p1, Punto p2 ){
    float dx = p1.x - p2.x ;
    float dy = p1.y - p2.y ;
    return sqrt( dx * dx + dy * dy );
}
```

Autoriferimento a strutture

- Una struttura può avere come membro un puntatore a una struttura
 - Anche puntatori alla medesima struttura!

```
/* primo esempio di
   lista */
struct punto {
    double x;
    double y;
    struct punto * next;
};

struct listaPunti{
    struct punto * primo;
}; /* continua... */
```

```
main() {
...
    struct listaPunti lis;
    struct punto pt1;
    struct punto pt2;
...
    lis.primo = &pt1;
    lis.primo->next = &pt2;
    pt2.next = NULL;
}
```

Tipo di enumerazione

- Insieme di costanti intere rappresentate da nomi (= identificatori)
 - Gli identificatori sono associati agli interi a partire da 0
- Definito con la keyword **enum**

```
enum seme {cuori, quadri, fiori, picche};  
/* cuori=0; quadri=1; fiori=2; picche=3 */  
enum seme miaCarta;  
int carteEstrate[4] = {0, 0, 0, 0};  
miaCarta = Estrai();  
/* estrazione casuale di una carta */  
if( miaCarta == cuori )  
    carteEstrate[cuori]++;  
/* conta le carte estratte per ogni seme */
```

Tipo di enumerazione

- Anche per i tipi enum si può compattare la dichiarazione con l'uso di typedef
- L'associazione degli identificatori agli interi può essere specificata dall'utente

```
typedef enum {lun=1, mar, mer, gio,  
             ven, sab, dom} giorno;  
/* lun=1; mar=2; mer=3;... */  
giorno day = mar;  
  
enum lettera {alfa=1, gamma=3, delta};  
/* alfa=1; gamma=3; delta=4; */
```

Tipo di enumerazione

```
/* Stampa i nomi dei giorni della settimana */
#include <stdio.h>

main() {
    typedef enum {lun, mar, mer, gio,
                 ven, sab, dom } giorno;
    /* lun=0; mar=1; mer=2;... */
    giorno g;
    char * nomeGiorno[] = {"Lunedì", "Martedì",
                           "Mercoledì", "Giovedì", "Venerdì", "Sabato",
                           "Domenica"};

    for(g=lun; g<=dom; g++)
        printf("%s \n", nomeGiorno[ g ]);
}
```

Allocazione statica della memoria

- Fino ad ora abbiamo visto esempi di **allocazione statica della memoria**
 - Quando si dichiara una variabile il compilatore riserva un certo numero di locazioni di memoria per contenere il dato associato a quella variabile
 - Le locazioni restano riservate per quella variabile per tutto il corso del programma

Array statici

- Il numero di elementi di un array va specificato in fase di dichiarazione
 - Infatti il compilatore deve sapere quanta memoria riservare al vettore
- Cosa faccio se non conosco in anticipo il numero di elementi che un vettore dovrà contenere?
 - Sono costretto a dichiarare un vettore con un numero sufficientemente grande di elementi
- Per i vettori l'allocazione statica può essere poco efficiente!

Allocazione dinamica della memoria

- Il linguaggio C consente di allocare la memoria anche in fase di esecuzione del programma (= a *run-time*)
 - L'operazione è effettuata mediante una funzione della standard library: `malloc()`

```
#include <stdlib.h>

void * malloc( size_t numeroDiBytes )
```

La funzione malloc()

- malloc() alloca un' area di memoria della dimensione specificata con il parametro numeroDiBytes
 - Si usa l' operatore sizeof per restituire il numeroDiBytes per un dato tipo
 - NOTA: sizeof è un operatore, non una funzione
 - Non occorre includere nessuna libreria
 - Il tipo restituito da sizeof è size_t
 - è in sostanza un tipo intero senza segno)

La funzione malloc()

- malloc() ritorna il puntatore alla prima locazione del blocco di memoria allocato dal sistema operativo
 - Il puntatore è di tipo void e dovrà essere effettuato un cast per restituire il puntatore al tipo desiderato
 - Se non c'è memoria disponibile viene restituito un puntatore NULL

La funzione free()

- Quando l'area di memoria non è più necessaria la si può restituire al sistema operativo (“rilasciare”) mediante la funzione free()
 - ATTENZIONE: per evitare *memory leak* ogni variabile allocata dinamicamente andrebbe rilasciata prima della fine del programma!

```
#include <stdlib.h>

void free( void * address )
```

Esempio

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int * p;
    p = (int *) malloc( sizeof(int) );
    if( p == NULL ) {
        printf("MEMORIA NON DISPONIBILE! \n");
        return 1;
    }
    *p = 100;
    printf("p = %x\n \ *p = %d\n", p, *p);
    free(p);
    return 0;
};
```

Memory leak

- **ATTENZIONE:** ogni variabile allocata dinamicamente andrebbe rilasciata prima della fine del programma!
- In caso contrario: pericolo di *memory leak*
 - Perdita del puntatore al blocco di memoria allocato
 - Il blocco di memoria non potrà più tornare al sistema operativo per tutta la durata del programma
 - Spreco di memoria

Esempio di memory leak

Memory leak

```
int * p;  
for(int i=0; i<10; i++){  
    p = (int *) malloc(sizeof(int));  
    *p = i;  
    printf(" %d\n", *p);  
}  
free(p);
```

*Alloco 10 blocchi
di memoria ma
rilascio solo l'ultimo*

No memory leak

```
int * p;  
for(int i=0; i<10; i++){  
    p = (int *) malloc(sizeof(int));  
    *p = i;  
    printf(" %d\n", *p);  
    free(p);  
}
```

Array dinamici

- Con l'uso della funzione `malloc()` è possibile allocare un array di elementi a run-time
 - Non è necessario conoscere la dimensione del vettore in fase di scrittura del programma
 - La dimensione del vettore dinamico sarà un dato di input del programma

Array dinamici:esempio

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int * p, dim;
    printf("Numero di elementi dell'array?\n");
    scanf("%d", &dim);

    p = (int *) malloc( dim * sizeof(int) );
    /* allocazione dell'array dinamico */

    if( p == NULL ) {
        printf("MEMORIA NON DISPONIBILE! \n");
        return 1;
    }

    /* continua... */
}
```

Array dinamici:esempio

```
/* ...continua */

for(int i = 0; i < dim; i++) {
    printf("Inserire elemento %d \n", i);
    scanf("%d", &p[i]);
}
/* inizializzazione dell'array dinamico */
...

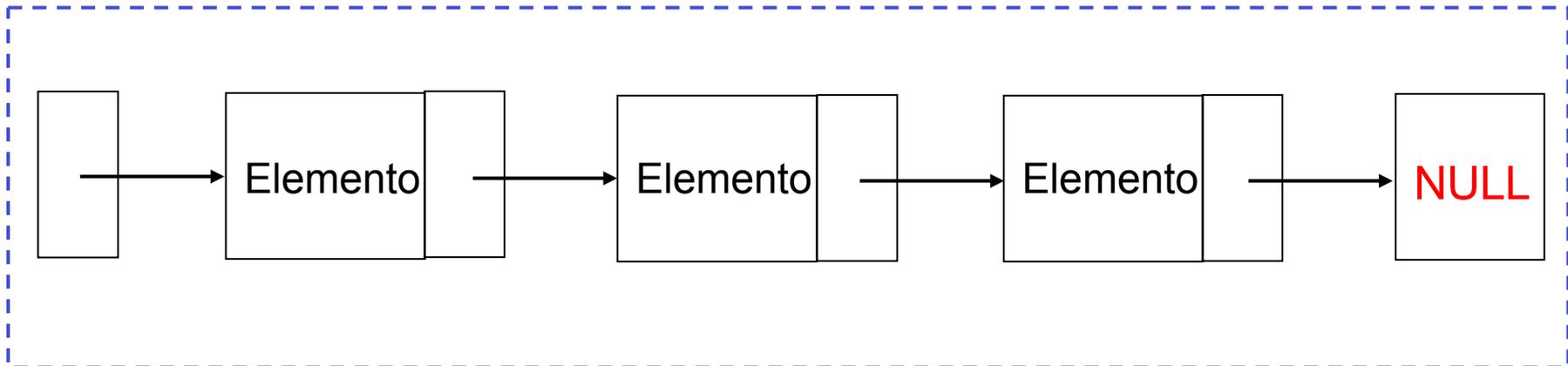
free(p);
/* libera la memoria allocata */

return 0;
};
```

Liste concatenate

- Una lista concatenata (*linked list*) è una collezione di elementi collegati mediante un puntatore

Lista concatenata



Liste concatenate

- Ogni elemento della lista deve contenere un puntatore al' elemento successivo
- La lista deve contenere un puntatore al primo elemento
- L' ultimo elemento punta a NULL
- Occorrono delle funzioni per manipolare una lista
 - Ad esempio funzioni per **inserire o cancellare un elemento**

Esempio: lista di studenti

- Definiamo anzitutto un elemento della lista
 - L'elemento deve contenere un puntatore a uno studente

```
struct studente{  
    int matricola, eta;  
    float media;  
    struct studente * next;  
};  
typedef struct studente Studente;
```

*Header file:
Studente.h*

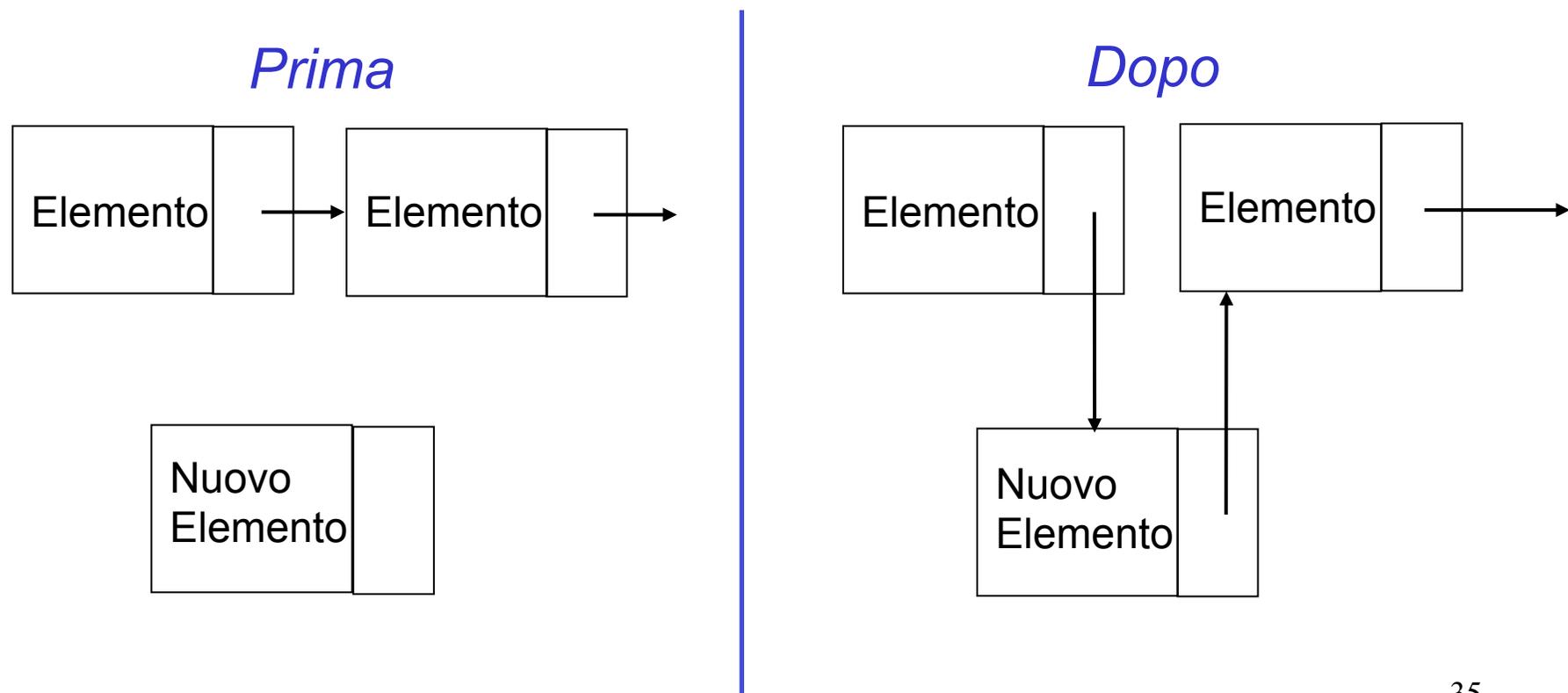
Esempio: lista di studenti

- **Definiamo la funzione che crea una lista**
 - Cioè crea una lista con un solo elemento (il primo) e ritorna il suo puntatore

```
Studente * creaListaStudenti () {  
    Studente * primo;  
    primo = (Studente * ) malloc( sizeof( Studente ) );  
    if (primo == NULL) {  
        printf("ERRORE: MEMORIA NON DISPONIBILE \n");  
        return 1;  
    }  
    primo->next = NULL;  
    return primo;  
}
```

Esempio: lista di studenti

- Definiamo la funzione che aggiunge un elemento a una lista dopo un certo elemento



Esempio: lista di studenti

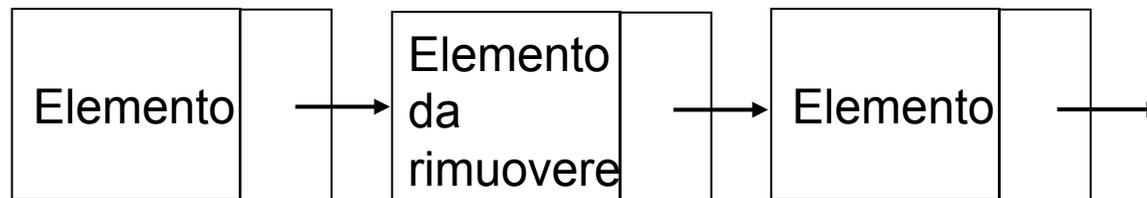
- Definiamo la funzione che aggiunge un elemento a una lista dopo un certo elemento

```
void inserisciStudente (Studente * st,  
                        Studente * nuovo) {  
  
    Studente * temp;  
    temp = st->next;  
    st->next = nuovo;  
    nuovo->next = temp;  
}
```

Esempio: lista di studenti

- Definiamo la funzione che rimuove uno studente dalla lista dopo un certo elemento

Prima



Dopo



Esempio: lista di studenti

- Definiamo la funzione che rimuove uno studente dalla lista dopo un certo elemento

```
void rimuoviStucente(Studente * elem) {  
    Studente * candToKill;  
    candToKill = elem->next;  
    elem->next = candToKill->next;  
    free(candToKill);  
}
```

Esempio: lista di studenti

- Definiamo la funzione che assegna le variabili di un elemento

```
void definisciStudiante(Studente * elem){
    if(elem!=NULL) {
        printf("Inserisci matricola, eta', media \n");
        scanf( "%d %d %f,", &(elem->matricola),
                &(elem->eta), &(elem->media) );
    }
}
```

Esempio: lista di studenti

- Esempio di main che usa la lista studenti

```
#include <stdio.h>
#include <stdlib.h>
#include "Studente.h"

Studente * creaListaStudenti();
void inserisciStudente(Studente * st,
                      Studente * nuovo);
void rimuoviStudente(Studente * elem);
void definisciStudente(Studente * elem);
/* prototipi delle funzioni per la gestione
   della lista */

int main() {
/* continua... */
```

Esempio: lista di studenti

- Esempio di main che usa la lista studenti

```
/* continua... */
int main(){
    int n;
    printf("Inserisci il numero degli studenti");
    scanf("%d", &n);

    Studente * primo = creaListaStudenti();
    /* crea la lista col primo elemento*/

    for(int i=1;i<n;i++) {
        Studente * nuovo = creaListaStudenti();
        inserisciStudente(primo, nuovo);
    }
    /* continua... */
```

Esempio: lista di studenti

- Esempio di main che usa la lista studenti

```
/* continua... */
  Studente * stud = primo;
  while( stud != NULL ) {
    definisciStudente( stud );
    stud = stud->next;
  }
/* definisce le variabili di ogni studente nella
   lista */
  return 0;
} /*fine del main */

... /* definizioni delle funzioni */
```