

Unità didattica: Implementazione C di una lista lineare [5-C]

Titolo: Implementazione C di una lista lineare: fondamenti

Argomenti trattati:

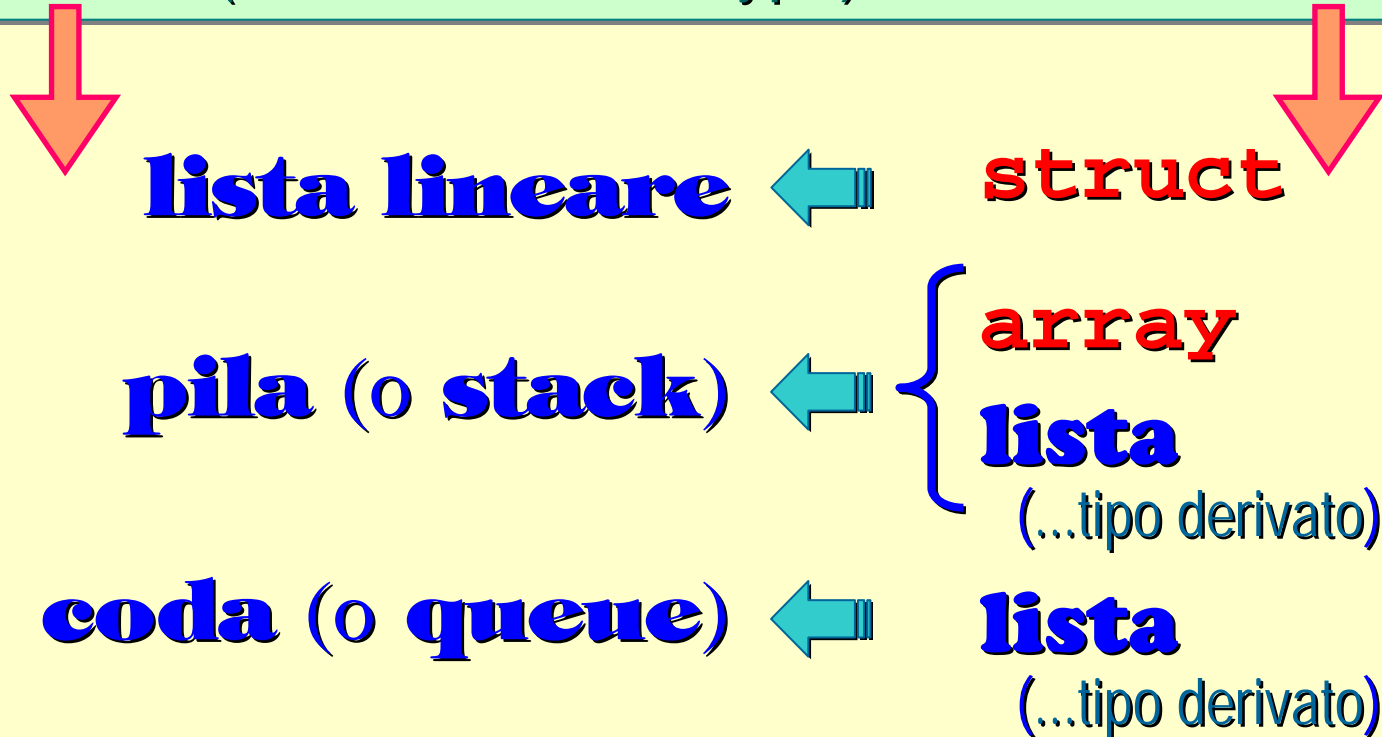
- ✓ Puntatore ad una struttura e notazione C per il puntatore ad un campo della struttura.
- ✓ Struttura C (`struct`) autoriferente statica e dinamica.
- ✓ Algoritmo per la visita di una lista lineare

Prerequisiti richiesti: variabili puntatore, allocazione dinamica, struttura dati lista lineare

Tipi strutturati (**pila**, **coda**, **lista**) in **C**

Per definire nel *linguaggio C* un tipo di dato strutturato lineare si ricorre quasi sempre al tipo primitivo **struct** per i singoli elementi della struttura (**nodi**) ed all'uso dei *puntatori* per la gestione dei *link* delle strutture dinamiche.

ADT (Abstract Data Type) ... mediante



Il tipo **lista lineare (linked list)** tramite struct

Il *tipo astratto record* (corrispondente in **C** al tipo **struct**) consente di introdurre come tipo derivato il tipo astratto *lista lineare* se si definisce un campo del record (in genere l'ultimo) come *puntatore allo stesso tipo di record*.



dichiara la struttura **PERSONA**

```

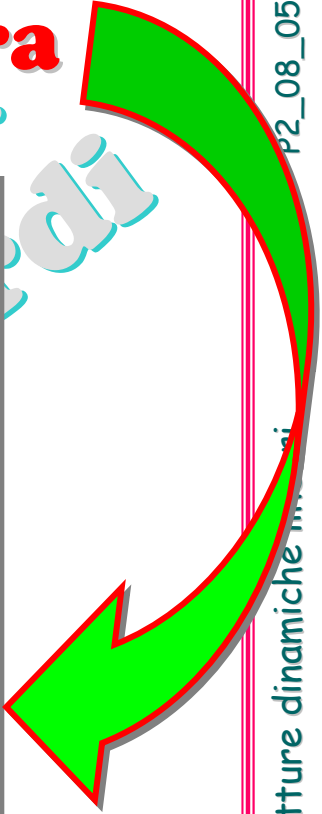
struct PERSONA
{
  char nome[20];
  struct PERSONA *p_next;
};
  
```

In C: puntatore ad una struttura

Es. 1

```
struct prenotazioni
{
    char nome[20];
    int aa,mm,gg;
};

struct prenotazioni utente;
struct prenotazioni *p_p;
p_p = &utente;
```



Attenzione alla precedenza tra operatori !!!

```
( *p_p ).nome
( *p_p ).nome[1]
( *p_p ).aa
```

```
*p_p.nome;
*p_p.nome[1]
*p_p.aa
```

Notazione puntatore a campo di struttura

invece di ...

`(*p_p).nome;`

è più chiaro ...

`p_p->nome;`

puntatore
alla struttura

`p_p`

utente

Rossi

02

04

25

“ -> ”

`p_p->nome`

`p_p->aa`

`p_p->nome[1]`

'R'

Es. 2

```
struct prenotazioni
{
    char nome[20];
    int aa, mm, gg;
};

struct prenotazioni utente[100];
struct prenotazioni *p_p;
p_p = utente+2;
```



utente[2].nome
utente[2].aa
utente[2].nome[1]

p_p->nome
p_p->aa
p_p->nome[1]

p_p

In C: **struttura autoriferente statica**

dichiara la struttura **PERSONA**

```
struct PERSONA
{
    char nome[20];
    struct PERSONA *p_next;
} e1_1, e1_2, e1_3;
```

dichiara
un campo della struttura
come puntatore
alla struttura

e1_1

Bianchi

e1_2

Rossi

e1_3

Verdi



... non serve a nulla !!!

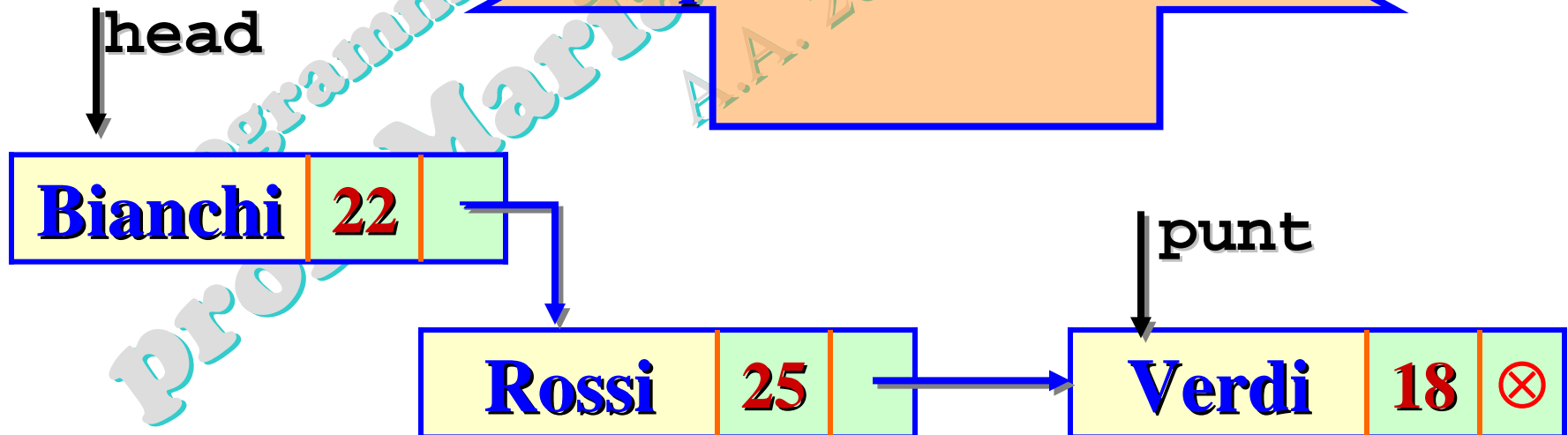
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{struct PERSONA
    {char nome[20];
      struct PERSONA *p_next;
    } el_1, el_2, el_3;
strcpy(el_1.nome, "Bianchi Roberto");
el_1.p_next=&el_2;
strcpy((el_2.nome, "Rossi Maurizio");
el_2.p_next=&el_3;
strcpy(el_3.nome, "Verdi Gianluca");
el_3.p_next=NULL;
printf("nome=%s, \tp_next=%d\n", el_1.nome, el_1.p_next);
printf("nome=%s, \tp_next=%d\n", el_2.nome, el_2.p_next);
printf("nome=%s, \tp_next=%d\n", el_3.nome, el_3.p_next);
}
```


In C: lista (struttura autoriferente dinamica)

dichiara la struttura **PERSONA**

```
struct PERSONA
{
    char nome[20];
    short eta;
    struct PERSONA *p_next;
} *head, *punt;
```

dichiara
dei puntatori alla struttura



versione semplificata senza functions

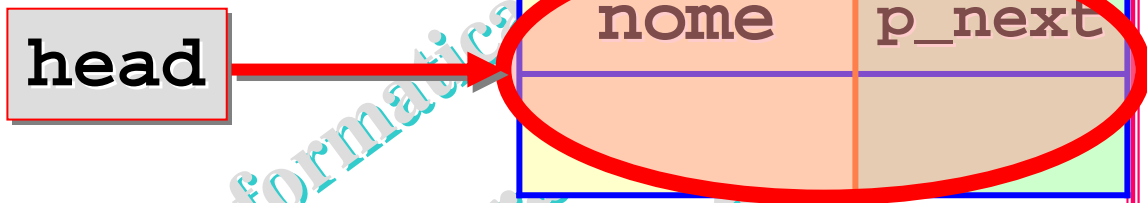
```
#include <string.h>
void main()
{struct PERSONA{char nome[20];
                short eta;
                struct PERSONA *p_next;} *head, *punt;
head = calloc(1, sizeof(struct PERSONA));
strcpy(head->nome, "Bianchi Roberto");
head->eta=22;
head->p_next = calloc(1, sizeof(struct PERSONA));
strcpy((head->p_next)->nome, "Rossi Maurizio");
(head->p_next)->eta=25;
(head->p_next)->p_next=calloc(1, sizeof(struct PERSONA));
strcpy(((head->p_next)->p_next)->nome, "Verdi Gianluca");
((head->p_next)->p_next)->eta=18;
((head->p_next)->p_next)->p_next=NULL;
punt = head;
while (punt->p_next != NULL)
{printf("nome=%s, \tp_next=%d\n", punt->nome, punt->p_next);
 punt = punt->p_next;}
printf("nome=%s, \tp_next=%d\n", punt->nome, punt->p_next);
}
```

alloca spazio per il 1° nodo e vi inserisce i dati

alloca spazio per il 2° nodo
e crea il link tra 1° e 2° nodo

dinamiche lineari

```
head=calloc(1, sizeof(struct PERSONA));
```

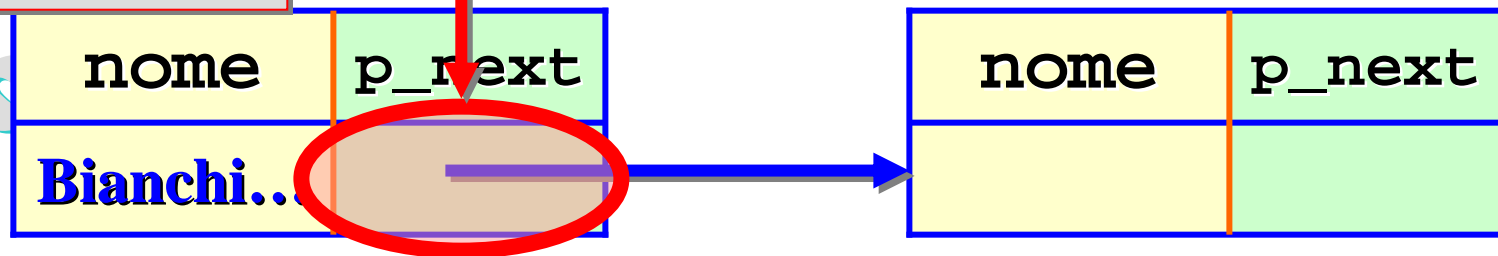


```
strcpy(head->nome, "Bianchi Roberto");
```



```
head->p_next=calloc(1, sizeof(struct PERSONA));
```

`head->p_next`

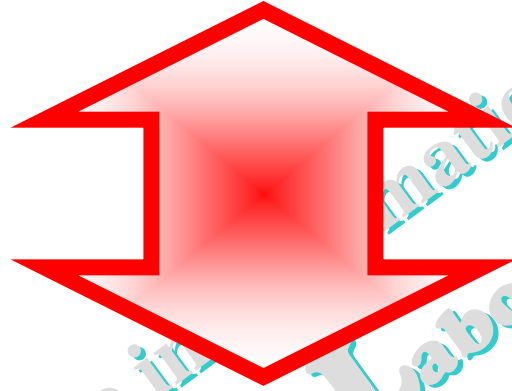


```
punt=head;  
while (punt->p_next != NULL)  
{  
    ...  
    punt = punt->p_next;  
};
```



Visita di una lista

`((head->p_next)->p_next)->p_next`



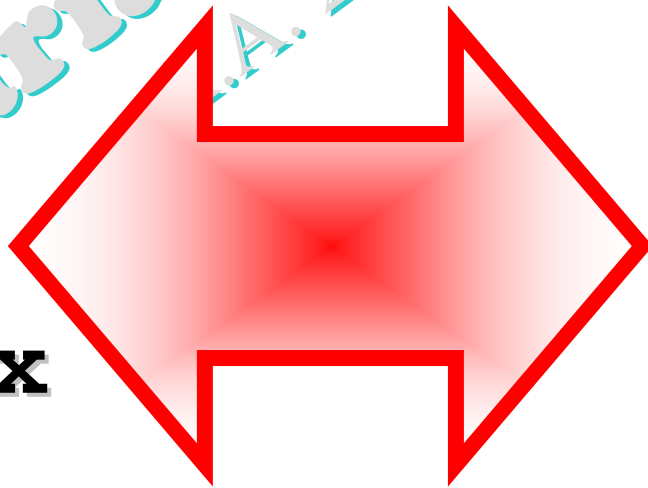
`head->p_next->p_next->p_next`

Gli operatori “.” e “->” sono associativi da sinistra a destra.

`r.pt1.x`

`rp->pt1.x`

`(rp->pt1).x`



`(r.pt1).x`

Gli **operatori di struttura** `.` e `->`, le **parentesi tonde** `(` e `)` per le chiamate a funzioni e le **parentesi quadre** `[` e `]` per gli indici di array hanno **priorità massima** sugli altri operatori.

~~`++px -> x`~~ \leftrightarrow `++(px -> x)`

~~`(++px) -> x`~~

`px++ -> x` \leftrightarrow `(px++) -> x`

`*px++ -> x`

\leftrightarrow `(*(px -> x))`
`px++`

Precedenza degli operatori

Livello

Operatori

1	fun () A [] -> .
2	! ~ ++ -- * _p &a (type) sizeof () + -
3	* / %
4	+ - aritmetici
5	<< >>
6	< <= > >=
7	== != relazionali
8	&
9	^ bitwise
10	
11	&&
12	booleani
13	? : condizionale
14	= += -= *= /= %= &= ^= = <<= >>= compatti
15	,