



# ***Corso di Programmazione a oggetti***

**Puntatori a funzione e ADT**

**a.a. 2013/2014**

**Francesco Fontanella**

---

# Puntatori a funzione

- Rappresentano l'indirizzo della prima istruzione di una funzione.
- L'impiego tipico: passaggio di una funzione come argomento ad un'altra funzione.
- Un puntatore a funzione può essere definito così:  
`tipo_restituito (*nome_puntatore) (lista_argomenti);`

# Puntatori a funzione: esempio

```
int (*pfun) (int x, int y);
```

- definisce il puntatore *pfun*
- Questo puntatore punta una funzione che ha due argomenti `int` passati per valore e restituisce un `int`.

- Per inizializzare un puntatore a funzione, è sufficiente assegnare il nome della funzione al puntatore.

```
int somma(int x, int y){  
    return x+y;  
}
```

```
int main() {  
    int (*pf)(int, int);  
  
    pf = somma;  
}
```

Definizione  
del puntatore

assegnazione  
del puntatore

- Per invocare la funzione puntata si scrive tra parentesi il nome del puntatore preceduto da un asterisco seguito dalla lista dei parametri effettivi:

```
int somma(int x, int y) {  
    return x + y;  
}
```

```
int main() {  
    int a=1,b=2,c;  
    int (*pf)(int,int);
```

```
    pf = somma;
```

```
    c = (*pf)(a,b);
```

```
}
```

Chiamata della  
funzione puntata

# Puntatori a funzione: Esempio d'uso

- Consideriamo l'esempio precedente dell'archivio di studenti.
- Si consideri di voler implementare la possibilità di ordinare l'elenco di studenti in diversi modi:
  - Per matricola
  - Per cognome
  - Per reddito
- Con i puntatori a funzione è possibile definire un'unica funzione di ordinamento, da usare per i tre modi di ordinamento

- Definisco una funzione di ordinamento che potremmo definire *generica* nel senso che il criterio di ordinamento non è fissato, ma è definito dalla funzione passata come parametro

```
void sort (Studiante v[], int n, bool (*comp)  
          (Studiante, Studiante))
```

```
// selectsort
void sort(Studente v[], int n, bool (*comp)
(Studente, Studente))
{
    int i, min, j;

    for(i=0; i < n ;i++) {
        min = i;
        for( j=i+1; j < n; j++) {
            if((*comp)(v[j], v[min]))
                min = j;
        }
        swap(v[i], v[min]);
    }
}
```

Chiamata alla  
funzione di confronto



```
// confronto le matricole
bool comp_matr(Studente s1, Studente s2)
{ return (s1.matr < s2.matr);
}

// confronto tra cognomi
bool comp_cognomi(Studente s1, Studente s2)
{
    int r;

    r choice= strcmp(s1.cognome < s2.cognome);
    return (r == 0);
}

// confronto le redditi
bool comp_redditi(Studente s1, Studente s2)
{ return (s1.red < s2.red);
}
```

```
// Dichiarazione delle funzioni
bool comp_matr(Studente s1, Studente s2);
bool comp_cognomi(Studente s1, Studente s2);
bool comp_redditi(Studente s1, Studente s2);

int main ()
{
    int choice, n;
    Studente *V;

    cout<<endl<<"come vuoi ordinare l'array?"<<endl;
    cout<<"1: matricola"<<endl;
    cout<<"2: cognome"<<endl;
    cout<<"3: reddito"<<endl;
    cin>>choice;
```

**SEGUE...**

```
switch(choice) {  
    case(1):  
        sort(V, n, comp_matr);  
        break;  
    case(2):  
        sort(V, n, comp_cognome);  
        break;  
    case(3):  
        sort(V, n, comp_redditi);  
        break;  
}
```

# Una esempio di ADT: Insieme

- Il tipo di dato astratto insieme consente di rappresentare collezioni (senza ripetizioni) di elementi di un tipo base
- Operazioni possibili:
  - *add(x)*
  - *remove(x)*
  - *member(x)*
  - *size()*
  - *empty()*
  - *clear()*

**DOMANDA**

Come rappresentiamo l'insieme?

```
struct Set{  
    void init(int l); // funzione di inizializzazione.  
    int    size();    // restituisce la cardinalità  
dell'insieme.  
    bool full(); // restituisce TRUE se è pieno.  
    bool empty(); // restituisce TRUE se è vuoto.  
    void clear(); // cancella tutti gli elementi  
    void add(TipoValue val); // aggiunge il valore val  
    void remove(TipoValue val); // rimuove l'elemento val  
    bool member(TipoValue val); // restituisce TRUE se val  
                                è presente.
```

**private:**

```
TipoValue *v;  
int len, n;  
elim(int pos);
```

```
};
```

**Qualificatore di accesso.**  
Rende inaccessibili  
all'esterno le variabili e  
Le funzioni che seguono

# ADT Insieme: implementazione

```
void Set::init(int l)
{
    v = new TipoValue[l];
    len = l;
    n = 0;
}
```

```
int Set::size()
{
    return n;
}
```

```
bool Set::full()
{
    return (n == len);
}
```

```
bool Set::empty()  
{  
    return (n == 0);  
}  
  
bool Set::member(TipoValue val)  
{  
    int i;  
  
    for (i=0; i < n; ++i)  
        if (v[i] == val)  
            return true;  
  
    return false;  
}
```

```
void Set::remove(TipoValue val)
{
    int i;
    bool found;

    i=0;
    found = false;
    while ((i < n) && (!found))
        if (v[i++] == val) {
            elim(i);
            found = true;
        }

    if (found)
        n--;

    return;
}
```

```
void Set::elim(int pos)
{
    int i, j;

    for (i=pos; j < n; ++i)
        v[i] == v[i+1];

    return;
}
```



```
void Set::clear()
{
    n = 0;
}

void Set::add(TipoValue val)
{
    if (!full()) {
        if (!member(val))
            v[n++] = val;
    }
    else
        cout<<endl<<"ERRORE: vettore pieno";

}
```

# Uso dell ADT Insieme

- Per usare l'ADT appena definita sono necessari tre passi:
  - Associare un tipo a TipoValue;
  - Includere il file insieme.h;
  - Ricompilare il file set.cpp

# Uso dell ADT Insieme: esempio

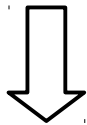
main.cpp

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

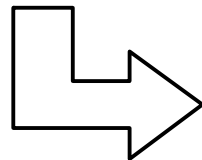
// Assegnazione di tipo
typedef Studente TipoValue;

// inclusione del file
#include "set.h"

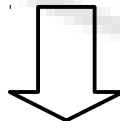
int main(){
    .
    .
    .
}
```



COMPILATORE



LINKER



eseguibile

set.cpp

```
#include <iostream>
#include "studente.h"

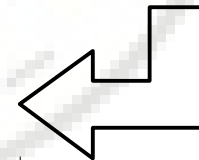
// Assegnazione di tipo
typedef Studente TipoValue;

// inclusione del file
#include "set.h"

void Set::init(int l)
{
    .
    .
    .
}
```



COMPILATORE



# Altri Esempi di ADT

- Pila
- Coda
- Lista



# ADT Pila

- L'accesso ai dati è del tipo **Last In First Out (LIFO)**

<b>PILA</b>	
<b>nome</b>	<b>semantica</b>
init	inizializzazione
push	inserisce un elemento in testa
pop	restituisce e rimuove l'elemento in testa
top	restituisce l'elemento in testa
full	la pila è piena?
size	numero di elementi nella pila
empty	la pila è vuota?
clear	cancella i dati contenuti

# ADT Pila: specifica

```
struct stack{

    void init(int l); // funzione di inizializzazione
    bool empty(); // controlla se la pila è vuota
    bool full(); // controlla se la pila è piena

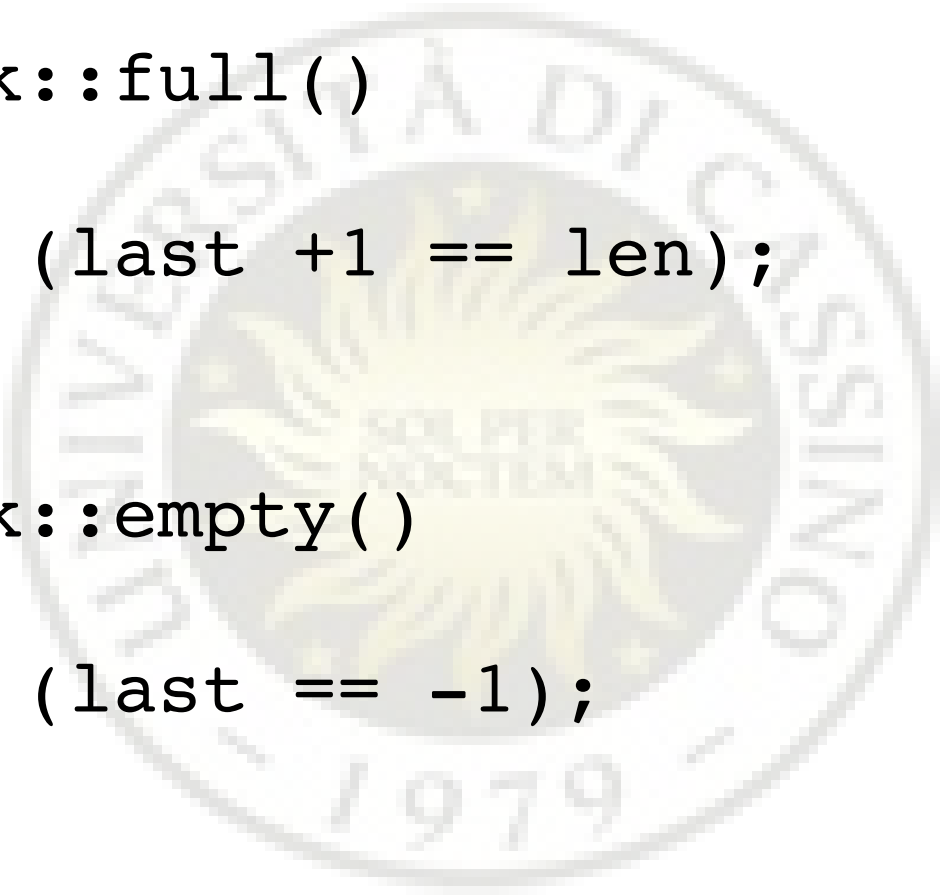
    TipoValue top(); // fornisce l'elemento in testa
    void push(TipoValue val); // inserimento
    TipoValue pop(); // fornisce e rimuove l'elemento in
    testa
private:
    TipoValue *v; // array per la memorizzazione
    int last; // punta all'ultimo elemento inserito
    int len;
};
```

# ADT Pila: implementazione

```
void stack::init(int l)
{
    v = new TipoValue[l];
    len = l;
    last = -1;

    return;
}

int stack::size()
{
    return last + 1;
}
```



```
bool stack::full()  
{  
    return (last +1 == len);  
}
```

```
bool stack::empty()  
{  
    return (last == -1);  
}
```



```
void stack::clear()
{
    last = -1;

    return;
}
```

```
TipoValue stack::top()
{
    if (last >= 0)
        return v[last];
    else {
        cout<<"ERRORE: stack vuoto!";
        exit(EXIT_FAILURE);
    }
}
```

```
void stack::push(TipoValue val)
```

```
{
```

```
    if (last < len)
```

```
        v[++last] = val;
```

```
    else cout<<"ERRORE: stack pieno";
```

```
    return;
```

```
}
```

```
TipoValue stack::pop()
```

```
{
```

```
    if (last >= 0)
```

```
        return v[last--];
```

```
    else {
```

```
        cout<<"ERRORE: stack vuoto!";
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
}
```

Prima si incrementa l'indice  
e poi si inserisce l'elemento

Prima si copia l'elemento  
e poi si decrementa l'indice

# ADT Coda

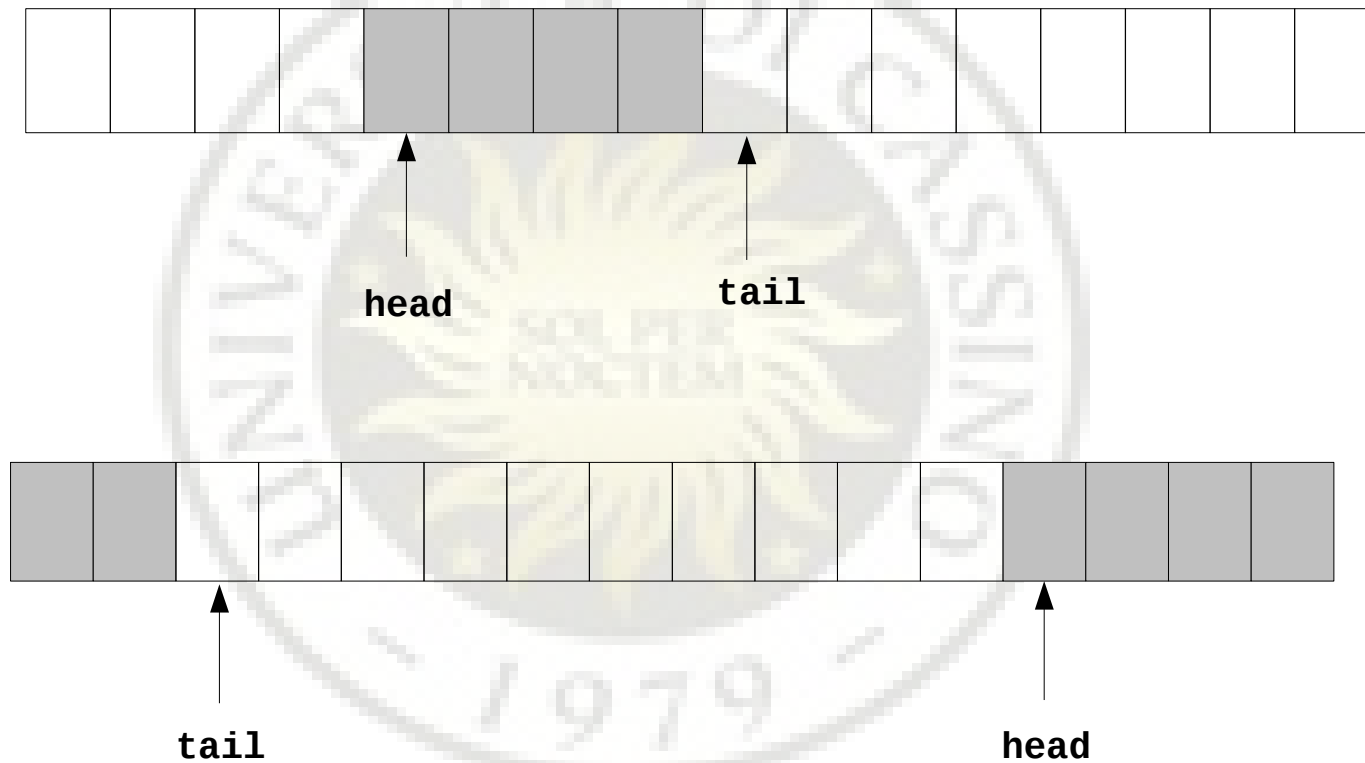
- L'accesso ai dati è del tipo **First In First Out (FIFO)**

CODA	
nome	semantica
init	inizializzazione
add	inserisce un elemento in coda
take	restituisce e rimuove l'elemento in testa
head	restituisce l'elemento in testa
size	Quanto è lunga la coda?
full	la coda è piena?
size	numeri di elementi nella coda
clear	cancella i dati contenuti

# ADT Coda: specifica

```
struct queue {  
  
    void init(int l); // funzione di inizializzazione  
    int size();       // restituisce l lunghezza della coda  
    bool full();      // controlla se la coda è piena  
    bool empty();     // controlla se la coda è vuota  
    void add(TipoValue val); // aggiunta di un elemento  
    TipoValue take();  // estrazione  
    TipoValue head();  // restituisce la testa  
  
private:  
    TipoValue *v;      // array di elementi  
    int h; // Punta alla testa della coda  
    int t; // Punta all'ultimo elemento della coda  
    int len; // capacità della coda  
};
```

# ADT coda: buffer circolare



# ADT Coda: implementazione

```
void queue::init(int l)
{
    v = new TipoValue[l];
    h = t = 0;
    len = l;

    return;
}
```


```
void queue::add(TipoValue val)
{
    if (!full())
        v[t] = val;
    else {
        cout<<"ERRORE: coda piena!";
        return;
    }

    // si incrementa la coda.
    t = (t + 1) % len;

    return;
}
```

Incremento in modulo len

```
TipoValue queue::take()  
{  
    TipoValue tmp;  
  
    if (!empty())  
        tmp = v[h]; // Si memorizza la testa  
    else {  
        cout<<"ERRORE: coda vuota!";  
        exit(EXIT_FAILURE);  
    }  
  
    // si incrementa la testa.  
    h = (h + 1) % len;  
  
    return tmp;  
}
```



Incremento in modulo len



```
int queue::size()
{
    if (t >= h)
        return (t - h);
    else return (len - (h - t));
}

bool queue::full()
{
    return (((t + 1) % len) == h);
}

bool queue::empty()
{
    return (t == h);
}
```

```
TipoValue queue::head()
{
    if (!empty())
        return v[h];
    else {
        cout<<"ERRORE: coda vuota!";
        exit(EXIT_FAILURE);
    }
}
```

# ADT Lista

- Una lista è una sequenza finita di elementi:

$$L = \langle a_0, a_1, \dots, a_n \rangle$$

LISTA	
nome	semantica
init	inizializzazione
insert(val, pos)	inserisce il valore val in posizione pos
remove(pos)	rimuove l'elemento in posizione pos
get(pos)	restituisce l'elemento in posizione
set(val, pos)	assegna il valore val all'elemento in posizione pos
size	Quanto elementi ci sono nella lista?
empty	la lista è vuota?
clear	cancella tutti i dati contenuti

# ADT Lista: specifica

```
struct List{  
    void init(); // funzione di inizializzazione.  
    int size(); // restituisce il numero di elementi.  
    void clear(); // svuota la lista  
    bool empty(); // lista è vuota?  
    void insert(TipoValue val, int pos); // inserisce il  
        valore val alla posizione pos  
    void remove(int pos); // rimuove l'elemento nella  
        // posizione pos  
    void set(TipoValue val, int pos); // assegna il valore  
        // val all'i-esima posizione.  
    TipoValue get(int pos); // restituisce il valore  
        // dell'elemento alla posizione pos.  
private:  
    TipoValue *v;  
    int n, len;  
    void ins(TipoValue val, int pos); // funzione ausiliaria  
        // di inserimento  
    void elim(int pos); // funzione ausiliaria per  
        // l'eliminazione  
};
```

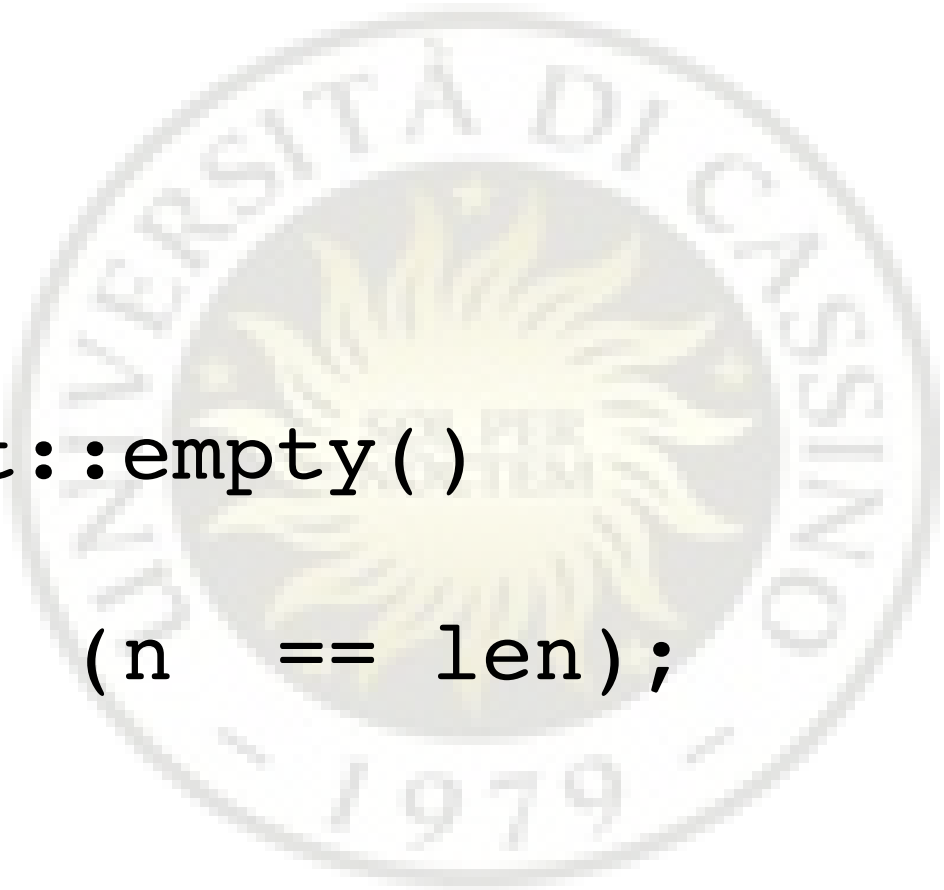
# ADT Lista: implementazione

```
const int MAX_ELEMENTS = 1000;

void List::init()
{
    len = MAX_ELEMENTS;
    v = new TipoValue[MAX_ELEMENTS];
    n=0;
}

int List::size()
{
    return n;
}
```

```
void List::clear()  
{  
    n = 0;  
}  
  
bool List::empty()  
{  
    return (n == len);  
}
```



```
void List::insert(TipoValue val, int pos)
{
    if (pos < 0 || pos > n) {
        cout<<"ERRORE! Valore pos errato!";
        return;
    }

    if (n < len)
        ins(val, pos);
    else {
        cout<<"ERRORE! Lista piena!!";
        return;
    }
    n++; // Si incrementa n.

    return;
}
```

```
Void List::ins(TipoValue val,
               int pos)
{
    int i;
    for (i=pos; i < n; ++i)
        v[i+1] == v[i];

    v[pos] = val;

    return;
}
```

```
void List::remove(int pos)
{
    if (pos < 0 || pos >= n) {
        cout<<"ERRORE! Valore pos errato!";

        return;
    }

    elim(pos);

    return;
}
```



```
void List::set(int pos, TipoValue val)
{
    if (pos < 0 || pos >= n) {
        cout<<"ERRORE! Valore pos errato!";

        return;
    }

    v[pos] = val;

    return;
}
```

```
TipoValue List::get(int pos)
{
    if (pos < 0 || pos == n) {
        cout<<"ERRORE! Valore pos errato!";
        exit(EXIT_FAILURE);
    }

    return v[pos];
}
```

# Rappresentazione delle liste

## ■ Rappresentazione sequenziale:

- Gli elementi della lista sono memorizzati uno dopo l'altro per mezzo di vettori

## ■ Rappresentazione collegata

- A ogni elemento si associa l'informazione che permette di individuare la posizione dell'elemento successivo tramite vettori

# Rappresentazione sequenziale

- Si utilizza un vettore per memorizzare gli elementi della lista uno dopo l'altro
- L'informazione necessarie sono:
  - indirizzo del vettore,
  - Numero di elementi

## ■ ESEMPIO

$\langle a, b, d, a, f, \rangle$

**vettore v**

a	b	d	a	f					
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]

# Rappresentazione sequenziale: vantaggi

- Accesso diretto agli elementi (tramite indice)
- L'ordine degli elementi è quello in memoria: non servono strutture dati particolari
- È semplice manipolare l'intera struttura

# Rappresentazione sequenziale: svantaggi

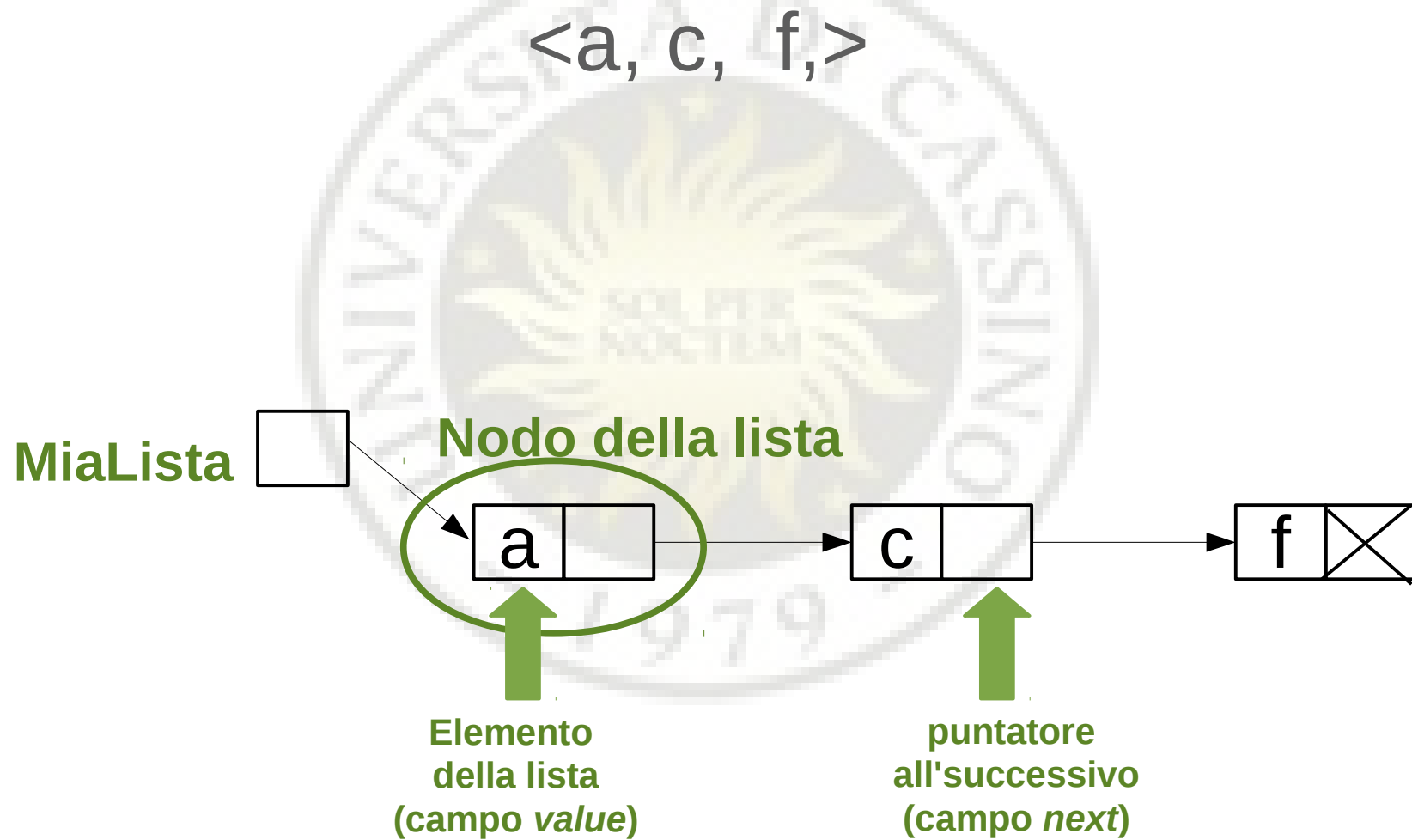
- Dimensioni fisse del vettore: dimensione massima della lista
- Le operazioni di inserimento e cancellazione sono molto costose, (bisogna spostare gli elementi che vengono dopo)

# Rappresentazione collegata

- Gli elementi sono memorizzati per mezzo di strutture chiamate **nodi**, dinamicamente allocate in memoria
- Ciascun nodo è una struttura di due campi:
  - Il valore dell'elemento a cui il Nodo si riferisce (campo *value*)
  - Il riferimento (puntatore) al Nodo associato all'elemento successivo
  - dell'insieme (campo *next*).



## ■ ESEMPIO



# Rappresentazione collegata

## ■ Vantaggi

- Non si ha più il problema della dimensione massima: nuovi elementi vengono aggiunti allocando dinamicamente un nuovo nodo
- Cancellazione ed inserimento efficienti: non richiedono più lo spostamento fisico di elementi ma solo aggiornamento di puntatori

## ■ Svantaggi

- Maggiore spazio occupato per ogni nodo. Questo overhead diventa trascurabile al crescere della dimensione dell'elemento
- Accesso non diretto agli elementi

# Liste collegate in C++

- In C++ un nodo può essere rappresentato per mezzo del tipo struct:

```
typedef struct nodo{  
    TipoValue  value;  
    nodo *next;  
}
```



## NOTA

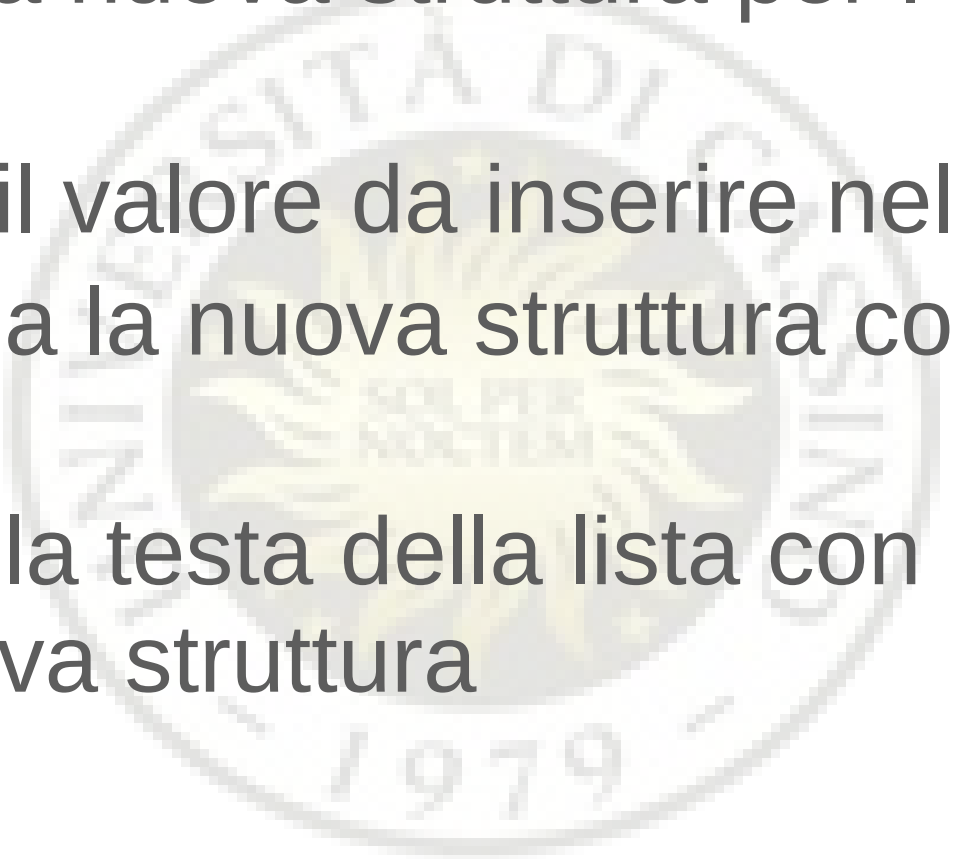
Il campo value, può essere di qualsiasi tipo: int, float, char o struct definite dall'utente

# Liste collegate: inizializzazione

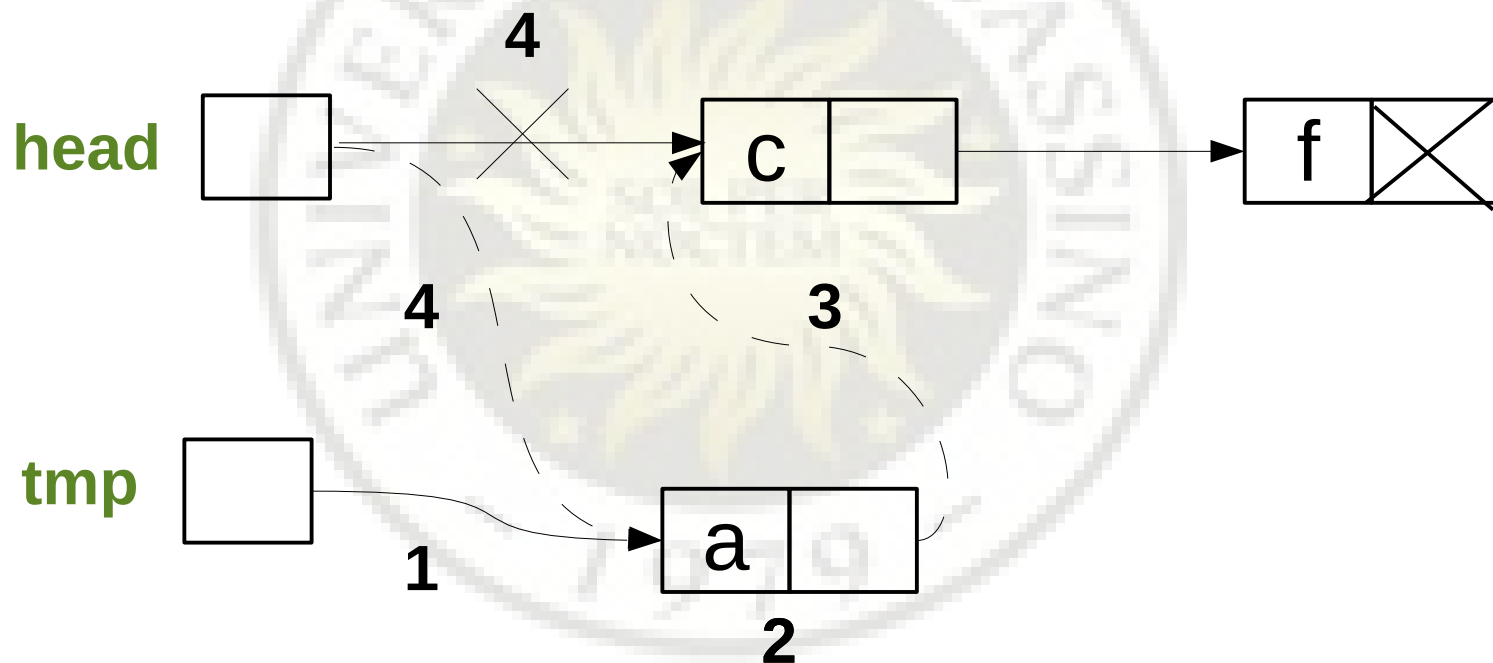
- Ogni lista è associata ad un puntatore iniziale che punta al suo primo elemento. Esso si inizializza con una lista vuota (puntatore nullo).
- Il primo elemento di una lista è detto *testa* (*head*).

`nodo *head = 0`

# Inserimento in testa

1. Alloca una nuova struttura per l'elemento da inserire
  2. Assegna il valore da inserire nella struttura
  3. Concatena la nuova struttura con la vecchia lista
  4. Aggiorna la testa della lista con l'indirizzo della nuova struttura
- 

# Inserimento in testa: rappresentazione grafica



# Inserimento in testa: codice

```
void ins_head(nodo* &l, TipoValue val)
{
    nodo * tmp;

    tmp = new TipoValue; //Passo 1

    tmp->value = val;      // Passo 2
    tmp->next = l;         // Passo 3

    l = tmp;               // Passo 4

    return;
}
```

Passaggio per riferimento

