

# Liste: realizzazioni in C++

**Algoritmi e Strutture Dati**

A.A. 18/19

Informatica

Università degli Studi di Bari "Aldo Moro"

Nicola Di Mauro

# Liste in C senza astrazione

```
#include <stdio.h>

/* Definizione tipo nodo */
typedef struct elemento_lista{
    int valore;
    struct elemento_lista *succ;
} nodo;
typedef nodo* lista; /* Definizione tipo lista di interi */

main(){
    lista iter, supp;
    lista L = NULL; /* creazione della lista */
    /* inserimento primo elemento */
    L = (lista) malloc (sizeof(nodo));
    L->valore = 1; L->succ = NULL;
    /* inserimento secondo elemento */
    iter = L;
    iter->succ = (lista) malloc (sizeof(nodo));
    iter->succ->valore = 2; iter->succ->succ = NULL;
    /* inserimento terzo elemento */
    iter = iter->succ;
    iter->succ = (lista) malloc (sizeof(nodo));
    iter->succ->valore = 3; iter->succ->succ = NULL;
    /* visualizzazione elementi */
    iter = L;
    while (iter != NULL){
        printf("%d ",iter->valore);
        iter = iter->succ;
    }
    /* eliminazione secondo elemento */
    iter = L->succ;
    L->succ = iter->succ;
    free(iter);
}
```

**lista-ex.c**

*Programma per l'inserimento, in una lista, di tre elementi, visualizzazione degli elementi, e rimozione del secondo elemento.*

Realizzazione con puntatori

- elementi di tipo intero
- non ci sono operatori
- il programma **funziona**

***E se volessimo una realizzazione con vettore?***

- cambia solo la realizzazione?
- cambia anche il programma?

**Programma vincolato alla realizzazione della lista!!**

# Liste in C senza astrazione /2

```
#include <stdio.h>

/* Definizione tipo nodo */
typedef struct elemento_lista{
    int valore;
    struct elemento_lista *succ;
} nodo;
typedef nodo* lista; /* Definizione tipo lista di interi */

main(){
    lista iter, supp;
    lista L = NULL; /* creazione della lista */
    /* inserimento primo elemento */
    L = (lista) malloc (sizeof(nodo));
    L->valore = 1; L->succ = NULL;
    /* inserimento secondo elemento */
    iter = L;
    iter->succ = (lista) malloc (sizeof(nodo));
    iter->succ->valore = 2; iter->succ->succ = NULL;
    /* inserimento terzo elemento */
    iter = iter->succ;
    iter->succ = (lista) malloc (sizeof(nodo));
    iter->succ->valore = 3; iter->succ->succ = NULL;
    /* visualizzazione elementi */
    iter = L;
    while (iter != NULL){
        printf("%d ", iter->valore);
        iter = iter->succ;
    }
    /* eliminazione secondo elemento */
    iter = L->succ;
    L->succ = iter->succ;
    free(iter);
}
```

**lista-ex.c**

```
#include <stdio.h>

/* Definizione tipo lista di interi */
typedef struct _lista {
    int elementi [3];
    int nelem;
} lista;

main()
{
    int i;
    /* creazione della lista */
    lista L;
    L.nelem = 0;
    /* inserimento primo elemento */
    L.elementi[nelem] = 1;
    L.nelem++;
    /* inserimento secondo elemento */
    L.elementi[nelem] = 2;
    L.nelem++;
    /* inserimento terzo elemento */
    L.elementi[nelem] = 3;
    L.nelem++;
    /* visualizzazione elementi */
    i = 0;
    while (i < L.nelem){
        printf("%d ", L.elementi[i]);
        i++;
    }
    /* eliminazione secondo elemento */
    L.elementi[1] = L.elementi[2];
    L.nelem--;
}
```

**lista-ex-v.c**

# Astrazione in C: liste con puntatori

```
#ifndef _LISTAP_H  
#define _LISTAP_H
```

```
typedef short int bool;  
typedef int tipoelem;
```

```
typedef struct _cella {  
    tipoelem elemento;  
    struct _cella* prec;  
    struct _cella* succ;  
} cella;
```

```
typedef cella* posizione;  
typedef cella* Lista;
```

```
void crealista(Lista);  
bool listavuota(Lista);  
tipoelem leggilista(posizione);  
void scrivilista(tipoelem, posizione);  
posizione primoLista(Lista);  
bool finelista(posizione, Lista);  
posizione succlista(posizione);  
posizione preclista(posizione);  
void inslista(tipoelem, posizione);  
void canclista(posizione);
```

```
#endif // _LISTAP_H
```

**listap.h**

# Astrazione in C: violazioni /1

```
#include "listap.h"
```

```
void crealista(Lista L){  
    tipoelem ElementoNullo;  
    L = (Lista) malloc(sizeof(cella));  
    L->succ = L; L->prec = L;  
}
```

```
bool listavuota(Lista L){ return ((L->succ == L) && (L->prec == L)); }  
posizione primoLista(Lista L){ return(L->succ); }  
posizione succlista(posizione p){ return p->succ; }  
posizione preclista(posizione p){ return p->prec; }  
bool finelista(posizione p, Lista L){ return (p==L); }  
tipoelem leggilista(posizione p) { return p->elemento; }  
void scrivilista(tipoelem a, posizione p){ p->elemento = a; }
```

```
void inslista(tipoelem a, posizione p){  
    posizione temp = (Lista) malloc(sizeof(cella));  
    temp->elemento = a;  
    temp->prec = p->prec;  
    temp->succ = p;  
    p->prec->succ = temp;  
    p->prec = temp;  
    p=temp;  
}
```

```
void canclista(posizione p){  
    posizione temp;  
    temp=p;  
    p->succ->prec = p->prec;  
    p->prec->succ = p->succ;  
    p=p->succ;  
    free(temp);  
}
```

**listap.c**

```
tipoelem leggilista(posizione)  
void scrivilista(tipoelem, posizione)  
posizione succlista(posizione)  
posizione preclista(posizione)  
void inslista(tipoelem, posizione )  
void canclista(posizione )
```

**Non rispettano le specifiche!!!!**

```
tipoelem leggilista(posizione p, Lista L){  
    return L.elementi[p];}
```

**Realizzazione con vettore**

**Questa realizzazione rispetta  
le specifiche**

# Astrazione in C: violazioni /2

```
#include "listap.h"
```

```
int main(){  
    Lista l;  
    crealista(l);  
    posizione indiceElemento = primoLista(l);  
    tipoelem a;  
    a = 1;  
    inslista(a,indiceElemento);  
    a = 2;  
    indiceElemento = succlista(indiceElemento);  
    inslista(a,indiceElemento);  
    a = 3;  
    indiceElemento = succlista(indiceElemento);  
    inslista(a,indiceElemento);  
    return 0;  
}
```

**testlista.c**

```
#include "listap.h"
```

```
int main(){  
    Lista l;  
    posizione p, indiceElemento;  
  
    crealista(l);  
    crealista(p);          /* violazione 1 */  
    indiceElemento = primoLista(l);  
    tipoelem a;  
    a = 1;  
    p = l;                 /* violazione 2 */  
    p->elemento = 12;      /* violazione 3 */  
    inslista(a,indiceElemento);  
    a = 2;  
    p = p->succ;           /* violazione 4 */  
    p->elemento = 31;      /* violazione 5 */  
    p = p->prec;          /* violazione 6 */  
    p = NULL;             /* violazione 7 */  
  
    return 0;  
}
```

**violazionilista.c**

# Realizzazione sequenziale con vettore

```
#ifndef _LISTAV_H
#define _LISTAV_H

// lunghezza massima della lista
const int DIMENSIONE = 1024;

// classe Lista
class Lista{
public:
    typedef int tipoelem;
    typedef int posizione;
    Lista();    // costruttore
    ~Lista();   // distruttore
    // operatori
    void creaLista();
    bool listaVuota() const;
    tipoelem leggiLista(posizione) const;
    void scriviLista(tipoelem, posizione);
    posizione primoLista() const;
    bool fineLista(posizione) const;
    posizione succLista(posizione) const;
    posizione predLista(posizione) const;
    void insLista(tipoelem, posizione);
    void cancLista(posizione);
private:
    tipoelem elementi[DIMENSIONE];
    int lunghezza;
};
#endif // _LISTAV_H
```

**listav.h (v0)**

- uso di un vettore per la memorizzazione degli elementi della lista
- vettori diversi per ogni istanza della classe
- vettore **elementi** monodimensionale che memorizza gli elementi della lista secondo la formula  $\text{posizione}(i) = i$
- variabile **lunghezza** che tiene traccia del numero di elementi presenti nella lista
- costante **DIMENSIONE** che tiene traccia della dimensione del vettore

# La classe Cella: Primo raffinamento

```
#ifndef _CELLALV_H  
#define _CELLALV_H
```

```
typedef int tipoelem;
```

```
// classe CellaLista  
class CellaLista{  
public:  
    CellaLista();    //costruttore  
    CellaLista(tipoelem);  
    ~CellaLista(){}; //distruttore  
    void scriviCella(tipoelem);  
    tipoelem leggiCella() const;  
    bool operator == (CellaLista);  
private:  
    tipoelem etichetta;  
};
```

```
#endif // _CELLALV_H
```

**cellalv.h (v0)**

Definizione della classe CellaLista  
Interfaccia

- classe CellaLista
  - parte pubblica
    - tipoelem
    - costruttori e distruttore
    - scriviCella e leggiCella
    - sovraccarico dell'operatore ==
  - parte privata
    - etichetta

**Come generalizzare il dato tipoelem?**



# La classe Libro

```
#ifndef _LIBRO_H
#define _LIBRO_H

#include <string>
#include <iostream>

using namespace std;

class Libro{
public:
    Libro();
    Libro(string);
    void setTitolo(string);
    string getTitolo() const;
    bool operator ==(Libro);
private:
    string titolo;
};

#endif // _LIBRO_H
```

**libro.h**

- classe libro
  - costruttori
  - metodi per scrittura e lettura
  - sovraccarico dell'operatore ==
  - titolo

**#include "libro.h"**

```
Libro::Libro(){ titolo = ""; }
```

```
Libro::Libro(string t){ setTitolo(t); }
```

```
void Libro::setTitolo(string t){
    titolo = t;
}
```

```
string Libro::getTitolo() const{
    return (titolo);
}
```

```
// sovraccarico dell'operatore ==
bool Libro::operator==(Libro l){
    return (getTitolo() == l.getTitolo());
}
```

**libro.cpp**

# La classe Cella

## Secondo raffinamento

```
#ifndef _CELLALV_H
#define _CELLALV_H
```

```
#include "libro.h"
```

```
typedef Libro tipoelem;
```

```
// classe CellaLista
class CellaLista{
public:
    CellaLista();    //costruttore
    CellaLista(tipoelem);
    ~CellaLista(){}; //distruttore
    void scriviCella(tipoelem);
    tipoelem leggiCella() const;
    bool operator == (CellaLista);
private:
    tipoelem etichetta;
};
```

```
#endif // _CELLALV_H
```

**cellalv.h**

```
#include "cellalv.h"
```

```
CellaLista::CellaLista() {}
```

```
CellaLista::CellaLista(tipoelem label){
    etichetta = label;
}
```

```
void CellaLista::scriviCella (tipoelem label){
    etichetta = label;
}
```

```
tipoelem CellaLista::leggiCella() const{
    return (etichetta);
}
```

```
bool CellaLista::operator==(CellaLista cella){
    return(leggiCella() == cella.leggiCella());
}
```

**cellalv.cpp**

# La classe Lista

## Terzo raffinamento

```
#ifndef _LISTAV_H
#define _LISTAV_H

#include "cellav.h"

// lunghezza massima della lista
const int DIMENSIONE = 1024;

// classe Lista
class Lista{
public:
    typedef int posizione;
    Lista();    // costruttore
    ~Lista();   // distruttore
    // operatori
    void creaLista();
    bool listaVuota() const;
    tipoelem leggiLista(posizione) const;
    void scriviLista(tipoelem, posizione);
    posizione primoLista() const;
    bool fineLista(posizione) const;
    posizione succLista(posizione) const;
    posizione predLista(posizione) const;
    void insLista(tipoelem, posizione);
    void canclLista(posizione);
private:
    CellaLista elementi[DIMENSIONE];
    int lunghezza;
};
#endif // _LISTAV_H
```

**listav.h**

# Implementazione della classe Lista

## Prima parte

**#include "lista.h"**

```
Lista::Lista(){ crealista(); }
```

```
Lista::~~Lista() {};
```

```
void Lista::creaLista() { lunghezza = 0; }
```

```
bool Lista::listaVuota() const {  
    return(lunghezza == 0);  
}
```

```
Lista::posizione Lista::primoLista() const{  
    return(1); // e quindi pos(1)=pos(n+1) se la lista è vuota (n=0)  
}
```

```
Lista::posizione Lista::succLista(posizione p) const{  
    if ( (0 < p) && (p < lunghezza+1)) // preconditione  
        return(p+1);  
    else return(p);  
}
```

```
Lista::posizione Lista::predLista(posizione p) const{  
    if ( (1 < p) && (p < lunghezza+1)) // preconditione  
        return(p-1);  
    else return(p);  
}
```

**listav.cpp**

# Implementazione della classe Lista

## Seconda parte

```
bool Lista::fineLista(posizione p) const{
    if ( (0 < p) && (p <= lunghezza+1)) // preconditione
        return( p == lunghezza+1);
    else return(false);
}

tipoelem Lista::leggiLista(posizione p) const{
    if ( (0 < p) && (p < lunghezza+1)) // preconditione
        return(elementi[p-1].leggiCella());
}

void Lista::scriviLista(tipoelem a, posizione p){
    if ( (0 < p) && (p < lunghezza+1)) // preconditione
        elementi[p-1].scriviCella(a);
}

void Lista::insLista(tipoelem a, posizione p){
    if ( (0 < p) && (p <= lunghezza+1)) { // preconditione
        for (int i=lunghezza; i>=p; i--) elementi[i] = elementi[i-1];
        elementi[p-1].scriviCella(a);
        lunghezza++;
    }
}

void Lista::cancLista(posizione p){
    if ( (0 < p) && ( p < lunghezza + 1)) // preconditione
        if (!listaVuota()){
            for (int i=p-1;i<(lunghezza-1);i++) elementi[i]=elementi[i+1];
            lunghezza--;
        }
}
```

# Funzioni di servizio

```
#ifndef _SERVIZIOLV_H
#define _SERVIZIOLV_H

#include <lista.h>

void stampaLista(Lista &);
void epurazioneLista(Lista &);
....

#endif // _SERVIZIOLV_H
```

## serviziolv.h

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

using namespace std;

void stampaLista(Lista &l){
    cout<<"[";
    Lista::posizione indice;
    for (indice = l.primoLista();
        (!!l.fineLista(indice)) && (indice < DIMENSIONE));
        indice = l.succLista(indice)){
        cout << l.leggiLista(indice).getTitolo();
        if (!!l.fineLista(l.succLista(indice))) cout << ", ";
    }
    cout<<"]\n";
}
.....
```

## serviziolv.cpp

Funzioni di servizio utente per le liste

- stampaLista: visualizzazione elem.
- epurazioneLista
- ...

# Test di Lista

```
#include <stdio.h>
#include <iostream>
#include <listav.h>
#include <servizioLista.h>

using namespace std;
int main(){
    Lista l;
    Lista::posizione indiceElemento = l.primoLista();
    Libro libro;

    libro.setTitolo("Primo");
    l.insLista(libro,indiceElemento=l.succLista(indiceElemento));
    libro.setTitolo("Secondo");
    l.insLista(libro,indiceElemento=l.succLista(indiceElemento));
    libro.setTitolo("Secondo");
    l.insLista(libro,indiceElemento=l.succLista(indiceElemento));
    libro.setTitolo("Quarto");
    l.insLista(libro,indiceElemento=l.succLista(indiceElemento));
    cout <<"\nL'attuale lista e': ";
    stampaLista(l);
    cout <<"\nOra inserisco l'elemento Dieci nella seconda posizione...\n";
    libro.setTitolo("Dieci");
    l.insLista(libro,l.succLista(l.primoLista()));
    cout << "\nLista inserita: " << endl;
    stampaLista(l);
    cout << "\nEpurazione lista: " << endl;
    epurazioneLista(l);
    stampaLista(l);
    return 0;
}
```

**testlista.cpp**

# Makefile: compilazione separata

# Macros

PROGRAM = listap

COMPILER = g++

FLAGS = -g

# Explicit rules

cellalp.o: libro.o cellalp.cpp

    \${COMPILER} \${FLAGS} -c cellalp.cpp

libro.o: libro.cpp

    \${COMPILER} \${FLAGS} -c libro.cpp

listap.o: cellalp.o libro.o listap.cpp

    \${COMPILER} \${FLAGS} -c listap.cpp

**Makefile**



# Overloading di operatori

- Operatori già sovraccaricati
  - +, -
- Il compilatore genera il codice appropriato
- Sovraccaricare un operatore
  - scrivere la definizione della funzione
  - il nome della funzione è costituito dalla parola chiave 'operator' seguito dal simbolo dell'operatore
    - operator+
- Usare l'operatore
  - per usare un operatore su una classe bisogna sovraccaricarlo (esclusi gli operatori = e &)
    - =: assegnamento delle componenti
    - &: restituisce l'indirizzo dell'oggetto

# Restrizioni in C++ /1

- Operatori sovraccaricabili
  - +, -, \*, /, %, ^, &, |
  - ~, !, =, <, >, +=, -=, \*=
  - /=, %=, ...
  - &&, ||, ==, --, ++, ...
  - [], (), new, delete, ....
- Operatori non sovraccaricabili
  - ., .\*, ::, ?::, sizeof

## Restrizioni in C++ /2

- Restrizioni di overloading
  - non è possibile cambiare l'arità
  - la precedenza e l'associatività di un operatore non possono essere cambiate
- Non si possono creare nuovi operatori
- Per i tipi primitivi non è possibile sovraccaricare gli operatori
  - non si può cambiare il modo in cui due interi vengono sommati

# Sovraccarico di operatori

- Operatori unari

```
class String {  
    public:  
        bool operator! () const;  
    ...  
};
```

- Operatori binari

```
class String {  
    public:  
        const String &operator+=(const String & );  
    ...  
};
```

-  $y += z$  è equivalente a  $y.operator+=(z)$

# Ereditarietà e classi astratte

- Ereditarietà

- una classe A può essere derivata da una classe B
  - la classe A è detta **classe derivata** (sotto-classe)
  - la classe B è detta **classe base** (super-classe)
- la classe derivata potrà accedere a tutti i campi della classe base definiti **protected**

- Classe astratta (interfaccia)

- contiene metodi (**funzioni virtuali**) senza implementazione

```
virtual int funzioneVirtuale(int x);
```
- una classe effettiva non contiene funzioni virtuali
- solo le classi effettive possono essere istanziate

# La classe astratta ListaLineare

```
template<class T>
class listaLineare
{
public:
    typedef int tipoelem;
    typedef int posizione;
    // distruttore
    virtual ~listaLineare() {}
    // operatori
    virtual void creaLista();
    virtual bool listaVuota() const;
    virtual tipoelem leggiLista(posizione) const;
    virtual void scriviLista(tipoelem, posizione);
    virtual posizione primoLista() const;
    virtual bool fineLista(posizione) const;
    virtual posizione succLista(posizione) const;
    virtual posizione predLista(posizione) const;
    virtual void insLista(tipoelem, posizione);
    virtual void canclLista(posizione);
}
```

- Tutte le classi derivate da una classe astratta sono delle classi astratte e quindi non possono essere istanziate a meno che non implementino tutte le funzioni virtuali della classe base.
- Richiedendo che ogni implementazione di ADT sia derivata da una classe astratta ci assicuriamo una implementazione completa e consistente.

# I template di classe

E' possibile comprendere che cosa sia una lista indipendentemente dal tipo di elementi che contiene.

Se si vuole istanziare una lista dobbiamo specificare necessariamente il tipo di dato contenuto.

I template di classe in C++ sono uno strumento per descrivere il concetto lista a un livello generale, tale da consentire di istanziare versioni specifiche.

I template di classe sono detti *tipi parametrici* perché hanno bisogno di uno o più parametri per generare l'istanza del template di classe desiderato.

Per generare una collezione di classi basta scrivere la definizione di un solo template di classe e, ogni volta che si ha bisogno di una nuova istanza specifica, è il compilatore che la genererà.

# Un template di classe **Lista**

Un template di classe **Lista**, per esempio, può essere la base per molte classi **Lista** come “**Lista** di **double**”, “**Lista** di **int**”, “**Lista** di **Libri**”, e così via.

```
template< class T >
class Lista {
    public:
        ...
    private:
        ...
    ...
}
```

Definizione del template di classe **Lista**

- template< class T > indica che si tratta di un template
- il parametro di tipo **T** indica il tipo di classe **Lista** da creare
- il tipo di elemento da memorizzare in Lista è menzionato genericamente come **T** in tutta l'istestazione della classe **Lista** e nelle definizioni delle funzioni membro



# Template

## Sintassi

### Definizione della classe template

```
template <typename variabile_tipo>
    class Nome_classe {
        caratteristiche
    }
```

### e della funzione membro template

```
template <typename variabile_tipo>
    tipo_restituito Nome_classe <variabile_tipo>::
        nome_funzione(parametri) constopt {
        istruzioni
    }
```

# Template

## Osservazioni

- Sono gestiti **staticamente** (cioè a livello di **compilazione**) e non comportano alcun costo aggiuntivo in fase di **esecuzione**.
- Sono utili per il programmatore che può scrivere del codice “**generico**” senza doversi preoccupare di differenziarlo in ragione della varietà dei **tipi** a cui tale codice va applicato.
- Vantaggiosi per creare **classi** con strutture identiche, ma diverse solo per i **tipi** dei **membri** e/o per i **tipi** degli **argomenti** delle **funzioni-membro**.
- La stessa **Libreria Standard del C++** mette a disposizione strutture precostituite di **classi template**, dette **classi contenitore** (**liste concatenate**, **mappe**, **vettori** ecc...) che possono essere utilizzate specificando, nella creazione degli **oggetti**, i valori reali da sostituire ai **tipi parametrizzati**.
- La definizione e l'implementazione di un template devono risiedere nello stesso file (non è possibile avere un header)

# Nuova realizzazione con vettore

- Stabilire qual'è il tipo di elemento del vettore
    - non è più necessario grazie ai template
  - Stabilire la dimensione del vettore
    - qual'è il numero massimo di elementi che una lista può ospitare?
      - difficile da stimare
    - Soluzione: aumentare dinamicamente la dimensione del vettore quando necessario
- Se **a** è il vettore iniziale
- definire un nuovo vettore con dimensione maggiore di **a**
  - copiare gli elementi di **a** nel nuovo vettore
  - cambiare il valore di **a** in modo che faccia riferimento al nuovo vettore

# Vettore a dimensione dinamica

```
template<class T>
void cambiaDimensione(T*& a, int vecchiaDim, int nuovaDim)
{
    if (nuovaDim < 0)
        throw illegalParameterValue("la nuova lunghezza deve essere >= 0");

    T* temp = new T[nuovaDim];
    int number;
    if (vecchiaDim < nuovaDim) then
        number = vecchiaDim;
    else
        number = nuovaDim;
    for (int i=0; i<number; i++)
        temp[i]=a[i];
    delete [] a;
    a = temp;
}
```

**array doubling**: quando viene utilizzato un vettore per la rappresentazione di una struttura la cui dimensione cambia dinamicamente, la lunghezza del vettore viene raddoppiata quando diventa pieno

# La nuova classe vetLista

```
#ifndef _VETLISTA_H  
#define _VETLISTA_H
```

```
template< class T >
```

```
class vetLista : public listaLineare<T>
```

```
{  
    public:  
        typedef T tipoelem;  
        typedef int posizione;  
        vetLista(int dim = 10); // costruttore  
        vetLista(const vetLista<T>&); // costruttore di copia  
        ~Lista(); //distruttore  
        // operatori  
        void creaLista();  
        bool listaVuota() const;  
        tipoelem leggiLista(posizione) const;  
        void scriviLista(tipoelem, posizione);  
        posizione primoLista() const;  
        bool fineLista(posizione) const;  
        posizione succLista(posizione) const;  
        posizione predLista(posizione) const;  
        void insLista(tipoelem, posizione);  
        void canclLista(posizione);  
    protected:  
        tipoelem* elementi; // vettore  
        int lunghezza; // lunghezza della lista  
        int dimensione; // dimensione del vettore  
};
```

# La nuova classe vetLista /2

// il costruttore crea un vettore la cui dimensione è dimIniziale che di default è 10

```
template< class T >
vetLista< T >::vetLista(int dim) {
    dimensione = dim;
    elementi = new T[dimensione];
    crealista();
}
```

/\* il costruttore di copia effettua una copia o clone di un oggetto. Questo costruttore viene invocato,  
\* per esempio, quando un oggetto viene passato per valore ad una funzione o quando una funzione  
\* restituisce un oggetto. \*/

```
template< class T >
vetLista< T >::vetLista(const vetLista<T>& Lista) {
    dimensione = Lista.dimensione;
    lunghezza = Lista.lunghezza;
    elementi = new T[dimensione];
    for (int i=0; i<Lista.dimensione; i++)
        elementi[i] = Lista.elementi[i];
}
```

```
template< class T >
void vetLista< T >::creaLista(){ lunghezza = 0; }
```

```
template< class T >
bool vetLista< T >::listaVuota() const { return(lunghezza == 0); }
```

...  
...

```
#endif // _VETLISTA_H
```

**vetLista.h**

## La nuova classe vetLista /2

- Istanziare un oggetto della classe vetLista
  - `vetLista L = new vetLista<int>(100);`
  - `vetLista<double> y(100);`
  - `vetLista<char> z;`
  - `vetLista<double> w(y);`
  - `vetLista<Libro> b(15);`

# Rappresentazione collegata circolare (con sentinella) realizzata mediante doppi puntatori (o simmetrica)

```
#ifndef _CELLALP_H  
#define _CELLALP_H
```

```
template< class T >
```

```
class Cella{
```

```
public:
```

```
typedef T tipoelem;
```

```
Cella() {}
```

```
Cella(const T& elemento){ this->elemento = elemento; }
```

```
void setElemento(tipoelem e){ elemento = e; }
```

```
tipoelem getElemento() const {return elemento; }
```

```
void setSucc(Cella *p){ succ=p; }
```

```
Cella * getSucc() const{ return succ;}
```

```
void setPrec(Cella *p) { prec=p;}
```

```
Cella * getPrec() const{ return prec;}
```

```
private:
```

```
tipoelem elemento;
```

```
Cella * prec;
```

```
Cella * succ;
```

```
};
```

```
#endif // _CELLALP_H
```

**cellalp.h**

Classe Cella

parte pubblica

- setElemento e getElemento
- setSucc e setPrec
- getSucc e getPrec

parte privata

- elemento
- prec
- succ



# La classe Lista

```
#ifndef _LISTAP_H
#define _LISTAP_H
```

```
#include "cella.h"
```

```
template<class T>
class cirLista{
public:
    cirLista();                // costruttore
    cirLista(const cirLista<T>&);    // costruttore di copia
    ~cirLista();                // distruttore
    /* posizione è un puntatore a cella */
    typedef Cella<T> * posizione;
    /* Prototipi degli operatori */
    void creaLista();
    bool listaVuota();
    tipoelem leggiLista(posizione);
    void scriviLista(tipoelem, posizione);
    posizione primoLista();
    bool fineLista(posizione);
    posizione succLista(posizione);
    posizione precLista(posizione);
    void insLista(tipoelem, posizione&);
    void canclLista(posizione &p);
private:
    posizione lista;
    //la lista è un puntatore ad oggetto Cella
};
```

## Classe Lista

parte pubblica

- posizione (puntatore a Cella)
- operatori

parte privata

- lista (puntatore a Cella)

# Implementazione della classe Lista

## Prima parte

**#include "listap.h"**

```
template< class T > circLista< T >::circLista() { crealista(); }
template< class T > const circLista< T >::circLista<T>& Lista) { /* da realizzare */ }
template< class T > circLista< T >::~~circLista(){
    while (lista->getSucc() != lista->getPrec()) canclista(lista->getSucc());
    delete lista;
}

template< class T > void circLista< T >::crealista(){
    tipoelem ElementoNullo;
    lista = new Cella;
    lista->setElemento(ElementoNullo);
    lista->setSucc(lista);
    lista->setPrec(lista);
    //la sentinella punta a se stessa
}

template< class T > bool circLista< T >::listaVuota() const{
    return ((lista->getSucc() == lista) && (lista->getPrec()==lista)); }

template< class T > circLista< T >::posizione circLista< T >::primoLista() const{
    return lista->getSucc(); }

template< class T > circLista< T >::posizione circLista< T >::succLista(Lista::posizione p) const{
    return p->getSucc(); }
template< class T > circLista< T >::posizione circLista< T >::precLista(Lista::posizione p) const {
    return p->getPrec(); }
template< class T > bool circLista< T >::fineLista(Lista::posizione p) const {
    return (p==lista); }
```

# Implementazione della classe Lista

## Seconda parte

```
template< class T > circLista< T >::tipoelem circLista< T >::leggiLista(posizione p) const{
    return p->getElemento();
}

template< class T > void circLista< T >::scriviLista(tipoelem a, posizione p){
    p->setElemento(a);
}

template< class T > void circLista< T >::insLista(tipoelem a, Lista::posizione &p){
    Lista::posizione temp;

    temp = new Cella;
    temp->setElemento(a);
    temp->setPrec(p->getPrec());
    temp->setSucc(p);
    (p->getPrec())->setSucc(temp);
    p->setPrec(temp);
    p=temp;
}

template< class T > void circLista< T >::cancLista(Lista::posizione &p){
    Lista::posizione temp;

    temp=p;
    (p->getSucc())->setPrec(p->getPrec());
    (p->getPrec())->setSucc(p->getSucc());
    p=p->getSucc();
    delete(temp);
}

#endif // _LISTAP_H
```

**circLista.cpp**

# Epurazione

Funzione di servizio **epurazione** (servizioLista.cpp)

```
void epurazioneLista(Lista &l){
    Lista::posizione p,q,r;

    p = l.primoLista();
    while (!l.fineLista(p)){
        q = l.succLista(p);
        while (!l.fineLista(q)){
            if (l.leggiLista(p) == l.leggiLista(q)){
                r = l.succLista(q);
                l.canclLista(q);
                q = r;
            }
            else
                q = l.succLista(q);
        }
        p=l.succLista(p);
    }
}
```

# Esercizi

- Realizzazione della struttura dati lista mediante cursori
- Ricerca in una lista lineare ordinata
- Fusione di liste ordinate
- Ordinamento di una lista