Progetto della classe MyList

- Progetto di una classe che permette di manipolare oggetti che rappresentano una "lista di interi"
- Due progetti: versione "bad", versione "good"
- La versione "bad"soffre del problema dell' *interferenza di memoria*
- La versione "good"risolve il problema definendo opportuni:
  - costruttore di copia
  - operatore di assegnamento
  - distruttore

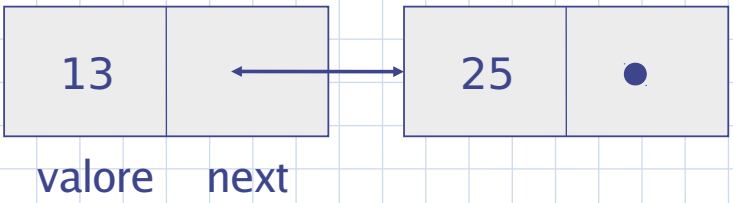
Il problema in esame

- Consideriamo una lista di interi, cioè una sequenza come:

  7 3 11 4 8 6
- Vogliamo progettare una classe in C++ in modo da poter utilizzare oggetti che rappresentano liste di interi
- Ci poniamo come obiettivo quello di una gestione dinamica della memoria:
  - la memoria utilizzata varia in funzione del numero di elementi nella lista
  - non ci sono vincoli sul numero di interi nella lista

L'elemento base della lista

```
class ListElement {
public:
  ListElement( int v, ListElement * n );

  void setValore (int);
  void setNext (ListElement *);
  int getValore();
  ListElement * getNext();

private:
  int valore;
  ListElement * next;
};
```



MyList (bad version)

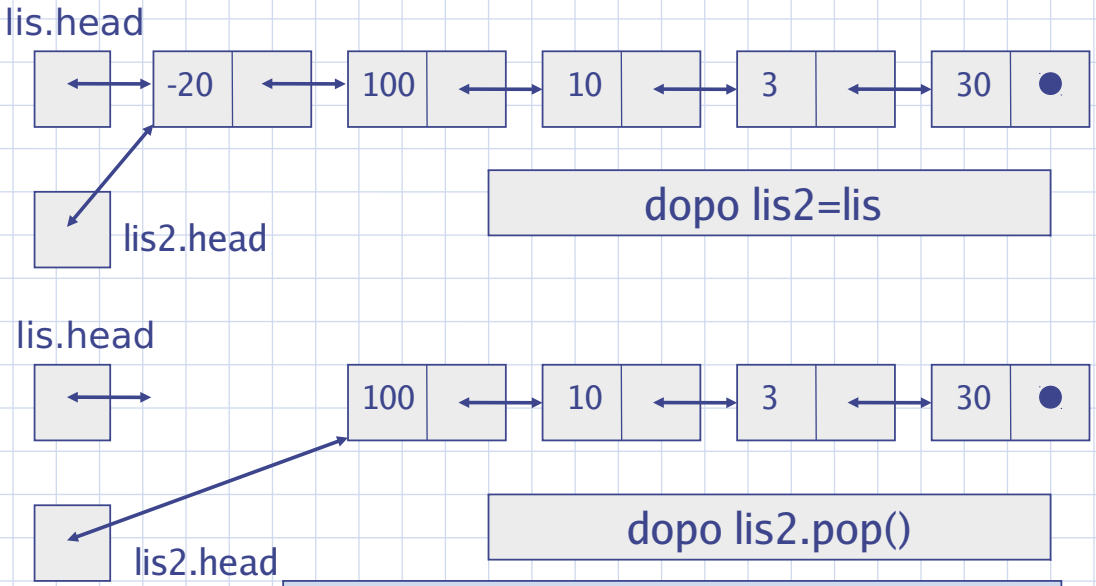
```
class MyList {
private:
  ListElement* head;
public:
  MyList();
  void insert(int);
  int pop();// estrazione
            // elemento in
            // testa alla lista
  void print();
};

MyList::MyList() {
  head = NULL;
}

void MyList::insert( int v ) {
  ListElement* tmp = new
    ListElement(v,head);
  head = tmp;
}
int MyList::pop() {
  if (head == NULL)  return 0;
  ListElement* tmp = head;
  head = head->getNext();
  int v = tmp->getValore();
  delete tmp;
  return v;
}
void MyList::print() {
  ListElement* tmp = head;
  while (tmp != NULL) {
    cout << tmp->getValore() << "\t";
    tmp = tmp->getNext(); }
  cout << endl;
}
```

Uso di oggetti MyList: *primo problema*

```
#include "MyList.h"
int main()
{    MyList lis;
     lis.insert(30);  lis.insert(3);  lis.insert(10); lis.insert(100); lis.insert(-20);
     lis.print();  // -20  100  10  3  30

     MyList lis2=lis; // questo NON DOVREBBE modificare l'oggetto lis

     lis2.pop();

     lis.print();  // 0  100  10  3  30  - lis è stata modificata !!
                   // in altri casi il metodo print() potrebbe non terminare !!
     ...
}
```

Perché lis è cambiata ??

Interferenza di memoria



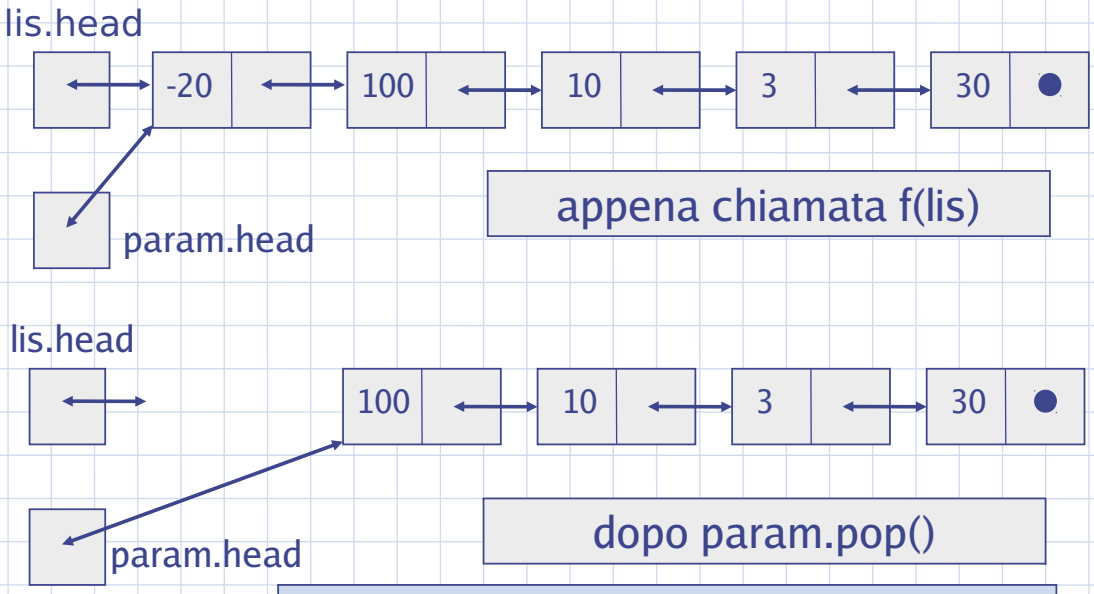
Per evitare questo effetto è necessario ridefinire l'**operatore di assegnamento** (operator=) in modo che esegua una copia profonda (elemento per elemento)

Uso di oggetti MyList: *secondo problema*

```
#include "MyList.h"
void f( MyList param )   // Att.ne: passaggio per valore!!
{  int v = param.pop();
   cout << "Nella funzione f(): il primo elemento e' " << v << endl; }

int main()
{   MyList lis;
    lis.insert(30);  lis.insert(3);  lis.insert(10); lis.insert(100); lis.insert(-20);
    lis.print();  // -20  100  10  3  30

    f(lis); // questa chiamata NON DOVREBBE modificare l'oggetto lis

    lis.print();  // 0  100  10  3  30
                  // in altri casi il metodo print() potrebbe non terminare !!
    ...
}
```

Perché lis è cambiata ??

Interferenza di memoria



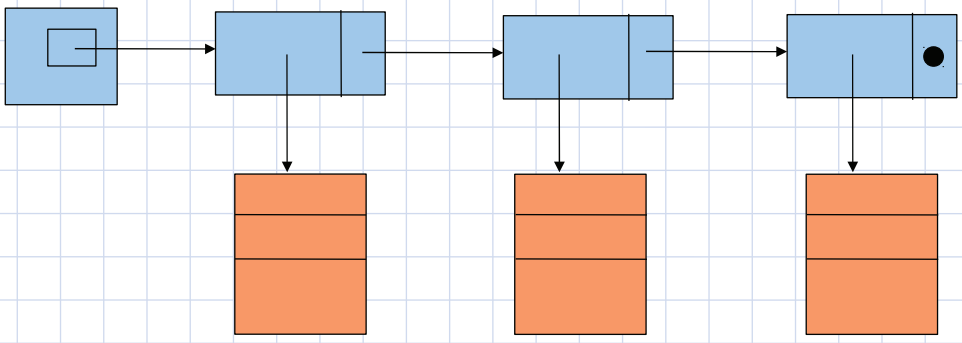
Per evitare questo effetto è necessario ridefinire il **costruttore di copia** in modo che esegua una copia profonda (elemento per elemento)

Tipi di dati astratti e concreti

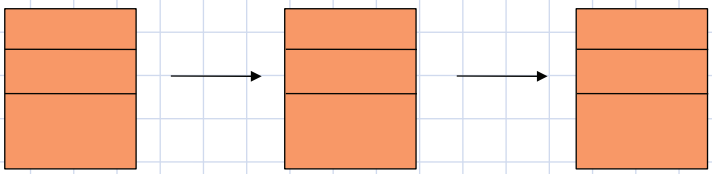
- Prima di trovare una soluzione ai problemi di MyList bisogna introdurre i concetti di tipo di dato **astratto (TDA)** e tipo di dato **concreto**.
- Ci sono due modi per considerare una lista concatenata:
  - Si può pensare alla sua realizzazione **concreta** con nodi collegati con puntatori
  - Oppure si può pensare al concetto **astratto** di lista come sequenza di dati ordinati che può essere attraversata con iteratori
- Analogamente si può pensare nei due modi a vettori, pile, code, alberi e in generale per ogni tipo di struttura dati.

Tipi di dati astratti e concreti

Visione concreta di una lista concatenata:

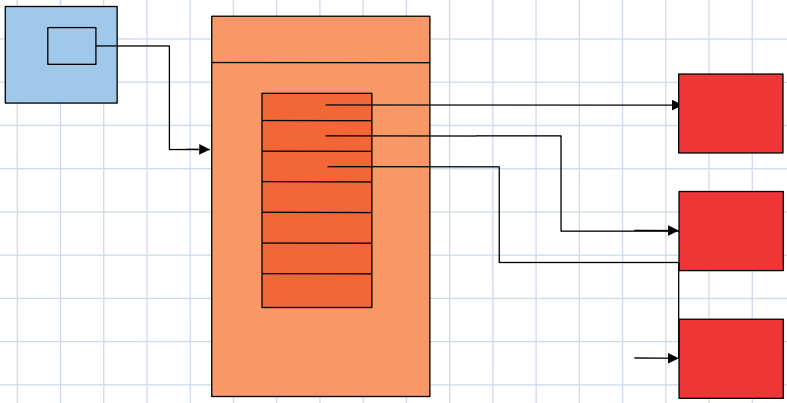


Visione astratta di una lista concatenata:

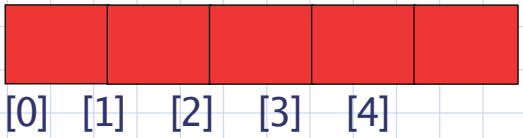


Tipi di dati astratti e concreti

Visione concreta di un vettore:

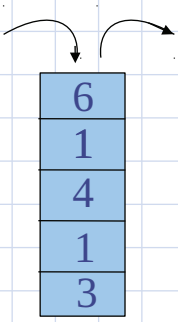


Visione astratta di un vettore:



Realizzazione di TDA pila

Pila: sequenza ordinata di elementi con politica di accesso LIFO:



Nei corsi precedenti si è caratterizzato il TDA pila con cinque operazioni primitive:

- Crea(p): inizializza una nuova pila p;
- Top(p): accede all'elemento affiorante della pila p;
- Push(x,p): aggiunge un elemento x alla pila p;
- Pop(p): rimuove un elemento dalla pila p;
- Vuota(p): compie il test di pila vuota.

Realizzazione di TDA pila

```
#ifndef PILA_H
#define PILA_H
typedef int elemento;        //elemento e' int per pila di interi
const int D= 100;

class Pila{
private:
   int top;                  //indice elemento top, se -1 pila vuota
   elemento elementi[D];
public:
    Pila();
    elemento Top();
    void Push(elemento);
    void Pop();
    bool Vuota();
};
#endif
```


Realizzazione di TDA pila

```
#include "pila.h"
#include <iostream>
using namespace std;

Pila::Pila(){  top = -1;}

elemento Pila::Top(){
 if (Vuota())
   cout<<"pila vuota"<<endl;
 else
    return elementi[top];
};

bool Pila::Vuota(){
 return top == -1;
};
```

```
void Pila::Push(elemento x){
 if (top == D-1)
   cout<<"pila piena"<<endl;
 else
    elementi[++top] = x;
};

void Pila::Pop(){
 if (Vuota())
   cout<<"pila vuota"<<endl;
 else
   top--;
}
```

Uso di oggetti Pila: *nessun problema*

```
##include "pila.h"
#include <iostream>
using namespace std;

int main(){
  Pila p;
  p.Push(30); p.Push(3); p.Push(10); p.Push(100); p.Push(-20);
  Pila p2 = p;

  p2.Pop();

  while (! p.Vuota()){  cout <<  p.Top() <<" ";   p.Pop(); }
  // -20  100  10  3  30
  while (! p2.Vuota()){  cout <<  p2.Top() <<" ";   p2.Pop(); }
  //  100  10  3  30 :nessun problema rispetto a MyList! Perche'?
}
```


MyList (good version)

```
class MyList {
private:
   ListElement* head;
   ListElement* deepCopy(ListElement*);  // effettua la copia elemento per elemento
   void del(ListElement*);  // dealloca TUTTI gli elementi della lista

public:
   MyList();
   MyList( const MyList & );   // costruttore di copia; usa deepCopy()
   ~MyList();  // distruttore; dealloca l'INTERA lista; usa del()
   void operator=(const MyList &);  // overloading operatore assegnamento;
                                    // usa deepCopy()
   void insert(int);
   int pop();
   void print();
};
```

MyList (good version)

```
MyList::~MyList() {
   del(head);
}

MyList::MyList(const MyList& orig) {
      head = deepCopy(orig.head);
    }

void MyList::del(ListElement* e) {
   if (e == NULL)
      return;
   del(e->getNext());
   delete e;
}
```

```
void MyList::operator=(const MyList& orig) {
   if (this == &orig)
      return;
   del(head);
   head = deepCopy(orig.head);
   return;
}

ListElement*
        MyList::deepCopy(ListElement* e) {
   if (e == NULL)
      return NULL;
   ListElement* tmp =
        new ListElement( e->getValore(),
                deepCopy(e->getNext()) );
   return tmp;
}
```


Problema: *uso operatori a cascata*

- Cosa fare se voglio poter scrivere quanto segue:

```
MyList l;
l.insert(30);  l.insert(3);
l.print();   // 3  30

MyList l1, l2;
l1 = l2 = l;  // uso a cascata dell'operatore =

l1.print();   // 3  30
l2.print();   // 3  30
```

- Notare che in C++ l'uso degli operatori a cascata è largamente diffuso

Soluzione

- Per risolvere il problema è sufficiente far restituire a operator=() l'oggetto di attivazione ...:

```
MyList & MyList::operator=(const MyList& orig) {
  if (this == &orig)
    return *this;
  del(head);
  head = deepCopy(orig.head);
  return *this;  // Nota: l'oggetto ritornato è ancora vivo
                 // dopo l'esecuzione del metodo
}
```

- In tal caso, l'esecuzione di l2 = l1 = l avviene nell'ordine:
  - si esegue l1= l e si ritorna l'oggetto di esecuzione (l1)
  - si esegue l2= l1 e si ritorna l'oggetto di esecuzione (l2, ma l'oggetto ritornato non viene più usato)