Sommario

- Collezioni:
  - Pile (stack)
  - Implementazione basata su array.
  - Esempio d'uso.
- Reference:
  - Tipi riferimento e puntatori.
- Costruttore per copia (copy constructor)
  - Problema del pointer aliasing
  - Esempio: linked list
  - Esempio: stack

Pile

- La *pila* (*stack*) è una collezione (un oggetto che raggruppa più dati in una singola struttura) con un comportamento specifico. Lo stack è un ADT.
- Lo stack è caratterizzato da:
  - Interfaccia pubblica.
  - Implementazione:
    - Strutture indicizzate (*array*): array di dimensione variabile.
    - Strutture collegate (*nodi*).
- Vi sono degli algoritmi che sfruttano il comportamento degli oggetti di tipo stack.

Pile

- *Una pila è una sequenza <a_1,…,a_n> di elementi dello stesso tipo, di cui solo l'ultimo elemento inserito, a_n, è visibile all'utente e la modalità di accesso è "ultimo elemento inserito, primo elemento rimosso" (LIFO: last in, first out).*
- È una struttura dati lineare a cui si può accedere soltanto mediante uno dei suoi capi per memorizzare e per estrarre dati.
- Una pila può essere descritta in termini di operazioni che ne modificano lo stato o che ne verificano lo stato: pertanto è un ADT.

Pile : interfaccia

```
class Stack{
…
public:
  Stack();
  ~Stack();
  void Push(char x);
  char Pop();
  char Top() const ;
  bool IsEmpty() const ;
  void Clear();
};
```

Inserisce l'oggetto `x` in cima alla pila.

Rimuove e restituisce l'oggetto in cima alla pila.

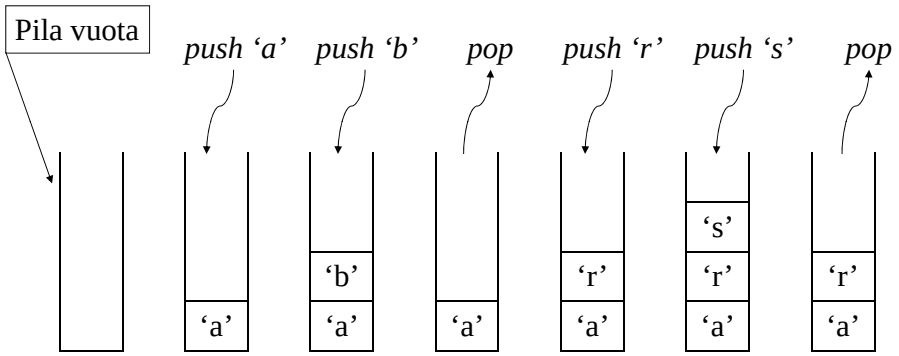
Restituisce l'oggetto in cima alla pila senza rimuoverlo.

Verifica se la pila è logicamente vuota.

Svuota la pila.

Pile

- Vediamo graficamente alcune operazioni su una pila.

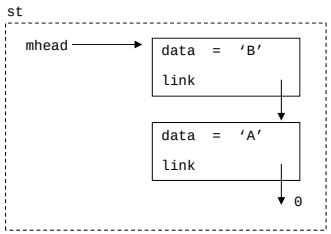


Pile: implementazione

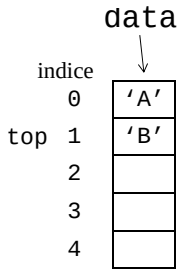
- Si consideri ora la *realizzazione* della pila. Si sono viste le operazioni per operare sulle pile, espresse nella definizione della classe `Stack`. E’ necessario fornire la definizione dei metodi della classe, ovvero dei metodi che operano sui dati della pila.
- L’implementazione può essere realizzata sfruttando due diversi tipi di strutture dati: strutture indicizzate (*array) o* strutture collegate *(nodi).*
- L’implementazione con *nodi* e’ argomento dell’esercitazione.

Pile: implementazione

PILE: implementazione mediante strutture collegate (*nodi*)



PILE: implementazione mediante array



Pile: implementazione con array

- L’implementazione con array consiste nell’usare un *array flessibile*, cioè un array che può *modificare dinamicamente* le sue *dimensioni*, perchè, in generale, non è noto a priori il numero di oggetti che una pila deve contenere.

Pile: implementazione con array

- Si realizza una struttura dati complessa utilizzandone una primitiva, l'array.
- Il pregio di tale implementazione è il *basso costo computazionale* per inserimenti ed estrazioni, *O(1)*, mentre il punto critico riguarda *l'uso della memoria.*
- Una sequenza di `Push()` può riempire la pila, mentre una sequenza di `Pop()` può lasciare un numero elevato di posizioni vuote, quindi nasce la necessità di ridimensionare opportunamente l'array.

Pile: implementazione con array

Nota sull'implementazione di un *array flessibile:*

- Allocare un nuovo array per ogni elemento da inserire ha un costo *O(n):* è necessario copiare tutti i valori dal vecchio array al nuovo. Quindi ogni operazione di inserimento risulta troppo onerosa, *O(n)*. La stessa cosa vale per ridurre la dimensione dell'array.
- È necessario utilizzare un accorgimento, la tecnica del *raddoppio e del dimezzamento*: il costo del ridimensionato dell'array può essere distribuito su più operazioni ottenendo un *costo medio costante*, *O(1)*.

Pile: *ArrayStack.h*

```
#ifndef ARRAYSTACK_H
#define ARRAYSTACK_H
class ArrayStack{
       int top;
       int size;
       int isize;
       char* data;
       void RaddoppiaArray();
       void DimezzaArray();
public:
       ArrayStack(int dim=5);
       ~ArrayStack();
       void Push(char x);
       char Pop();
       char Top() const;
       bool IsEmpty() const;
       void Clear();
};
#endif
```

metodi privati

Pile: *ArrayStack.cpp*

```
ArrayStack::ArrayStack(int dim)
{
       cout<<"ArrayStack() Dimensione iniziale "<<dim<<endl;
       data=new char[dim];
       size=dim;
       isize=dim;
       top=-1;
}
ArrayStack::~ArrayStack()
{
       cout<<"~ArrayStack()"<<endl;
       delete[] data;
       size=0;
       top=-1;
}
```

Pile: *ArrayStack.cpp*

```
void ArrayStack::Push(char x)
{
       if(top==size-1)
              RaddoppiaArray();
       data[++top]=x;
}
char ArrayStack::Pop()
{
       if(IsEmpty())
              return '\0';
       if ((top<size/4) && (top>=isize/4) )
              DimezzaArray();
       return data[top--];
}
```

E' necessario raddoppiare l'array

E' necessario dimezzare l'array

Pile: *ArrayStack.cpp*

```
void ArrayStack::RaddoppiaArray(){
   cout<<"Raddoppiamento Array---Nuova dimensione "<<size*2<<endl;
   char* tmp=new char[size*2];
   for(int i=0; i<size; i++)
       tmp[i]=data[i];
   delete[] data;
   size=size*2;
   data=tmp;
}
void ArrayStack::DimezzaArray(){
   cout<<"Dimezzamento Array---Nuova dimensione "<<size/2<<endl;
   char* tmp=new char[size/2];
   for(int i=0; i<=top; i++)
       tmp[i]=data[i];
   size=size/2;
   delete[] data;
   data=tmp;
}
```


Pile: *ArrayStack.cpp*

```
char ArrayStack::Top() const
{
       if(IsEmpty())
              return '\0';
       else
              return data[top];
}
bool ArrayStack::IsEmpty() const
{
       return top==-1;
}
void ArrayStack::Clear()
{
       delete[] data;
       size=isize;
       data = new char[size];
       top=-1;
}
```


Pile: *test.cpp*

```
void Test1()
{
       ArrayStack s1(5);
       for(int i=0; i<26; i++)
              s1.Push('a'+i);
       while(!s1.IsEmpty())
              cout<<s1.Pop()<<" ";
       cout<<endl;
       bool res=s1.IsEmpty();
       if(res)
              cout<<"Lo stack e' vuoto"<<endl;
       else
              cout<<"Lo stack non e' vuoto"<<endl;
       cout<<endl;
}
```

Pile: *test.cpp*

Una possibile uscita:

```
ArrayStack() Dimensione iniziale 5
Raddoppiamento Array --- Nuova dimensione 10
Raddoppiamento Array --- Nuova dimensione 20
Raddoppiamento Array --- Nuova dimensione 40
z y x w v u t s r q p o n m l k Dimezzamento Array--- Nuova dimensione 20
j i h g f Dimezzamento Array--- Nuova dimensione 10
e d c Dimezzamento Array--- Nuova dimensione 5
b a
Lo stack e' vuoto

~ArrayStack()
```


Pile: prestazioni

- Le operazioni `Push()` e `Pop()` vengono eseguite a costo costante *O(1*), essendo realizzate con un array.
- L'inserimento di un elemento in una *pila piena* richiede l'assegnazione di maggiore memoria e gli elementi del vettore pieno sono copiati in quello nuovo. Quindi inserire elementi nel caso peggiore richiede un costo *O(n).* Tuttavia si può pensare di avere ancora *costo costante in media*: tale operazione avviene in media dopo *n inserimenti*, quindi il *costo distribuito* per ogni operazione è *O(1*). Lo stesso vale per l'estrazione di un elemento.

Pile: prestazioni

- Nell'implementazione basata sull'uso di *strutture collegate,* i dati sono memorizzati in nodi, ogni nodo ha un riferimento al precedente.
- In questo caso non vi sono problemi di memoria: si alloca solo il numero necessario di nodi. Ogni operazione ha costo *O(1).*
- Ogni inserimento provoca la creazione di un oggetto nodo. Ogni estrazione distrugge un oggetto nodo.

Pile: esempio – bilanciamento delle parentesi

- Un'applicazione delle pile è l'identificazione dei *delimitatori corrispondenti* in un'espressione, che *è* un esempio significativo perché questa attività fa parte di qualsiasi *compilatore.*
- Vediamo un esempio:
  - Uso corretto dei delimitatori:
    `b+(c-d)*(e-f)`
  - Mancata corrispondenza:
    `b+(c-d)*(e-f))`

Pile: esempio – bilanciamento delle parentesi

- L'*algoritmo* legge un carattere dalla stringa che rappresenta l'espressione e se si tratta di un elimitatore iniziale, '(' o '[' o '{', lo memorizza in una pila. Quando viene trovato un delimitatore finale, ')' o ']' o '}', esso viene confrontato con quello estratto dalla pila: se corrispondono l'analisi continua, altrimenti termina segnalando un errore.
- Vediamo una possibile implementazione.

Pile: esempio – bilanciamento delle parentesi

```
bool BilanciamentoParentesi(string expr){
    ArrayStack st(256);
    for (unsigned int i = 0; i < expr.size(); i++){
        char ch = expr[i];
        switch (ch){
           case '(': case '[':case '{':
                st.Push(expr[i]); break;
           case ')':case ']':case '}':
                if(st.IsEmpty())
                        return false;
                char ctr = st.Pop();
                if (!(ctr == '(' && ch == ')' || ctr == '[' &&
                                ch == ']'|| ctr == '{' && ch == '}'))
                        return false;
                break;
        }
    }
    if (!st.IsEmpty())
                return false;
        else
                return true;
}
```


Pile: esempio – bilanciamento delle parentesi

Vediamo l'analisi della seguente espressione: 2 + [(3 - 2) * (5 - a) + b]

Stack:	Letto: 2	+[(3-2)*(5-a)+b]
Stack:	Letto: +	[(3-2)*(5-a)+b]
Stack:	Letto: [	(3-2)*(5-a)+b]
Stack: [	Letto: (	3-2)*(5-a)+b]
Stack: (	Letto: 3	-2)*(5-a)+b]
Stack: (	Letto: -	2)*(5-a)+b]
Stack: (	Letto: 2	)*(5-a)+b]
Stack: (	Letto:)	*(5-a)+b]
Stack: [	Letto: *	(5-a)+b]
Stack: [	Letto: (	5-a)+b]
Stack: (	Letto: 5	-a)+b]
Stack: (	Letto: -	a)+b]
Stack: (	Letto: a	)+b]
Stack: (	Letto:)	+b]
Stack: [	Letto: +	b]
Stack: [	Letto: b	]
Stack: [	Letto:]	

Ok!

Pile: esempio – bilanciamento delle parentesi

Vediamo l'analisi della seguente espressione: 2 + [(3 - 2)) * (5 - a) + b]

Stack:	Letto: 2	+[(3-2))*(5-a)+b]
Stack:	Letto: +	[(3-2))*(5-a)+b]
Stack:	Letto: [	(3-2))*(5-a)+b]
Stack: [	Letto: (	3-2))*(5-a)+b]
Stack: (	Letto: 3	-2))*(5-a)+b]
Stack: (	Letto: -	2))*(5-a)+b]
Stack: (	Letto: 2	))*(5-a)+b]
Stack: (	Letto:)	)*(5-a)+b]
Stack: [	Letto:)	*(5-a)+b]

No

Pile: esempio – bilanciamento delle parentesi

Vediamo l'analisi della seguente espressione: 2 + [(3 - 2) * (5 - a) + b] }

Stack:	Letto: 2	+[(3-2)*(5-a)+b]}
Stack:	Letto: +	[(3-2)*(5-a)+b]}
Stack:	Letto: [	(3-2)*(5-a)+b]}
Stack: [	Letto: (	3-2)*(5-a)+b]}
Stack: (	Letto: 3	-2)*(5-a)+b]}
Stack: (	Letto: -	2)*(5-a)+b]}
Stack: (	Letto: 2	)*(5-a)+b]}
Stack: (	Letto:)	*(5-a)+b]}
Stack: [	Letto: *	(5-a)+b]}
Stack: [	Letto: (	5-a)+b]}
Stack: (	Letto: 5	-a)+b]}
Stack: (	Letto: -	a)+b]}
Stack: (	Letto: a	)+b]}
Stack: (	Letto:)	+b]}
Stack: [	Letto: +	b]}
Stack: [	Letto: b	]}
Stack: [	Letto:]	}
Stack:	Letto: }	

No

Pile: esempio – bilanciamento delle parentesi

```
void Test4()
{
	bool res;
	res = BilanciamentoParentesi("2+[(3-2)*(5-a)+b]}");
	if(res)
		cout<<"Espressione bilanciata!"<<endl;
	else
		cout<<"Errore bilanciamento!"<<endl;
}
```

Una possibile uscita:

```
ArrayStack() Dimensione iniziale 256
~ArrayStack()
Errore bilanciamento!
```


TIPI RIFERIMENTO

- Un riferimento (*reference)* agisce come un nome alternativo per un oggetto: permette la manipolazione indiretta di un oggetto in maniera simile all'uso di un puntatore, ma senza richiedere l'uso della sintassi dei puntatori:

```
double d = 3.3;
double &ref = d;
ref=5; // d vale 5 e ref vale 5
```

Notare l'uso di &

- Un riferimento deve essere sempre inizializzato

```
double &r; //errore: compile-time
```


TIPI RIFERIMENTO

- Una volta definito, un riferimento *non* può essere riferito ad un altro oggetto:

```
double x =0;
ref = x; // d vale 0 e ref vale 0
x=123; // d vale 0 e ref vale 0
```

- Tutte le operazioni sui riferimenti sono applicate effettivamente agli oggetti cui si riferiscono, compreso l'operatore *indirizzo-di:*

```
double *p= &ref;
*p=33; // d vale 33 e ref vale 33
```

TIPI RIFERIMENTO

- Le due differenze fondamentali tra puntatori e riferimenti sono: (1) un riferimento deve sempre riferirsi ad un oggetto; (2) l'assegnamento di un riferimento ad un altro cambia il valore dell'oggetto riferito e non il riferimento stesso.
- L'utilizzo più frequente dei riferimenti si ha come *parametri formali* di una funzione: un altro modo per implementare il *passaggio per riferimento* oltre all'uso dei puntatori. Utili per (i) passare come argomenti oggetti molto grandi o (ii) modificare i valori degli argomenti.

TIPI RIFERIMENTO

```
void swap_ptr(int *v1, int *v2){
     int temp=*v2;
     *v2 = *v1;
     *v1 = temp;
}
void swap_ref(int &v1, int &v2){
     int temp=v2;
     v2 = v1;
     v1 = temp;
}
```

riferimenti

uso semplificato senza *

TIPI RIFERIMENTO

```
int main(){
     int a=1, b=9;
     swap_ptr( &a, &b);
     cout<<"a="<<a<<" b="<<b<<endl;

     a=1; b=9;
     swap_ref( a, b);
     cout<<"a="<<a<<" b="<<b<<endl;
     return 0;
}
```

Chiamata con variabili senza &, ma a e b sono modificate!

Una possibile uscita:

```
a=9 b=1
a=9 b=1
```


TIPI RIFERIMENTO

- Prima di dereferenziare un puntatore si dovrebbe verificare che punti effettivamente ad un oggetto, questo non è necessario con un riferimento.
- Se all'interno di una funzione un parametro deve riferirsi a più oggetti o a nessuno allora è necessario usare un puntatore.
- Se si desidera che un parametro riferimento non sia modificato nella funzione, lo si deve dichiarare const.
- I parametri riferimento consentono di implementare in modo efficiente gli *operatori sovraccaricati* mantenendone intuitivo l'uso.

COSTRUTTORE PER COPIA

- L'inizializzazione di un oggetto di una classe con un altro oggetto della stessa classe viene chiamata *inizializzazione di default membro a membro.*
- Tale inizializzazione avviene automaticamente se non è fornito un costruttore esplicito (*copy-constructor).*
- L'operazione eseguita è quella di copia di ogni attributo.

COSTRUTTORE PER COPIA

- L'inizializzazione di un oggetto con un altro della sua classe avviene nelle seguenti situazioni:
  - Inizializzazione esplicita.

```
Point p1(1,2);
Point p2(p1);
```

  - Passaggio di un oggetto come argomento di una funzione.
  - Passaggio di un oggetto come valore di ritorno di una funzione.

COSTRUTTORE PER COPIA

- Il comportamento di default non è adeguato quando:
  - Un attributo è un puntatore a memoria allocata nello *heap*, in tal caso viene copiato solo l'indirizzo e non la memoria: quindi si hanno due oggetti che condividono la stessa area di memoria (*pointer aliasing).*
  - Un attributo deve avere un valore diverso per ogni oggetto (per es. il numero di conto bancario).
- In tali casi è necessario definire un costruttore per copia che implementi la corretta semantica di inizializzazione della classe.

Esempio: ArrayStack

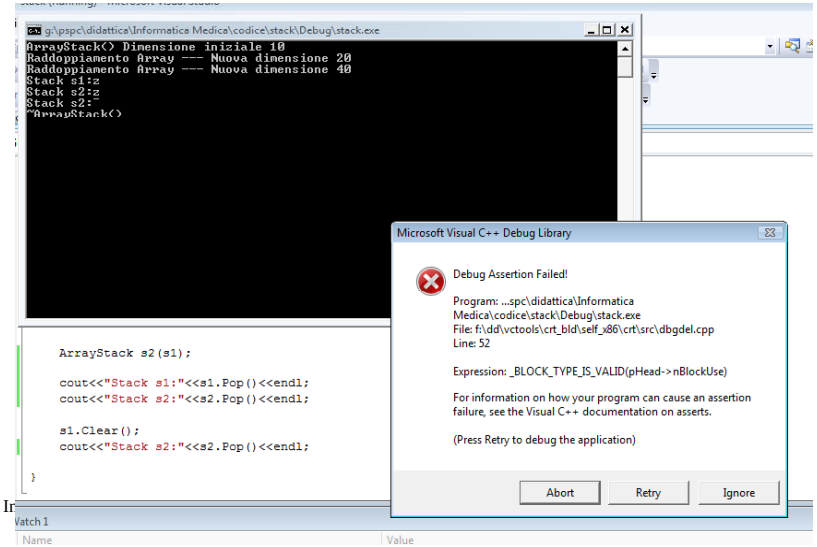
Vediamo un esempio in cui il comportamento di default del costruttore per copia non e' adeguato:

```
void TestCopyConstructor{
       ArrayStack  s1(10);
       for(int i=0; i<26; i++)
              s1.Push('a'+i);

       ArrayStack s2(s1);

       cout<<"Stack s1:"<<s1.Pop()<<endl;
       cout<<"Stack s2:"<<s2.Pop()<<endl;

       s1.Clear();
       cout<<"Stack s2:"<<s2.Pop()<<endl;
}
```

costruttore per copia

Esempio: ArrayStack

Una possibile uscita (comportamento errato!):



Esempio: ArrayStack – pointer aliasing

- Il comportamento di default del costruttore per copia genera il problema *dell'aliasing dei puntatori* (pointer aliasing).
- Viene effettuata una copia membro a membro di tutti gli attributi della classe, compreso il puntatore `char* data`, ma non viene allocata una nuova area di memoria.
- Pertanto i due oggetti `s1` e `s2` condividono la stessa area di memoria, generando comportamenti inattesi e/o errori run-time.

Esempio: linked list

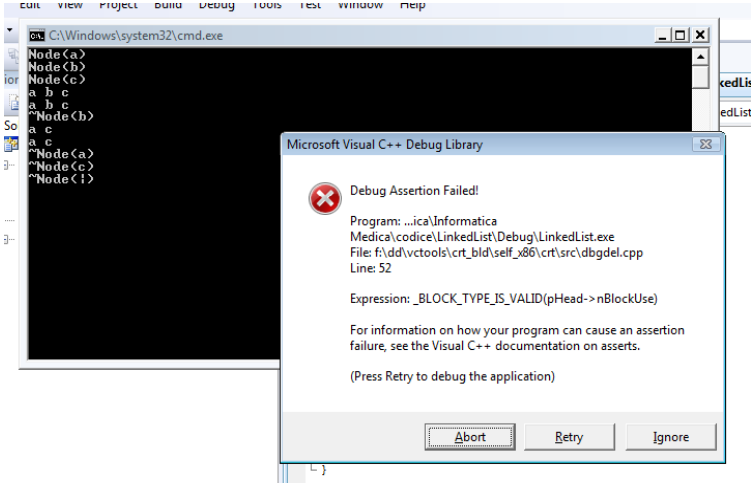
Vediamo un altro esempio in cui il comportamento di default del costruttore per copia non e' adeguato:

```
void TestCopyConstructor()
{
        LinkedList l1;
        l1.Add('a');
        l1.Add('b');
        l1.Add('c');
        l1.Print();
        LinkedList l2(l1);
        l2.Print();
        l1.Remove('b');
        l1.Print();
        l2.Print();
}
```

costruttore per copia

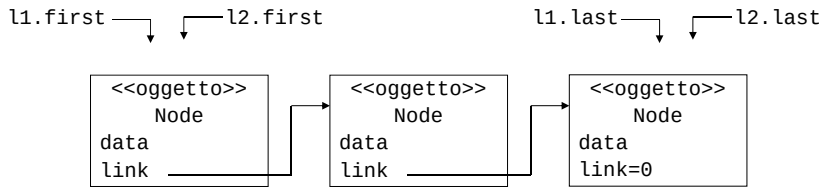
Esempio: linked list

Una possibile uscita (comportamento errato!):



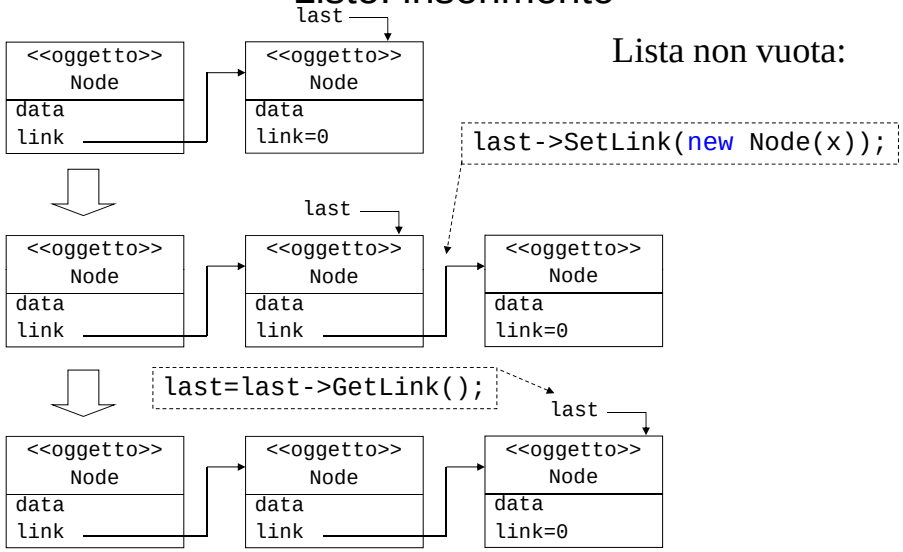
Esempio: ArrayStack – pointer aliasing

• Anche per la classe LinkedList il comportamento di default del costruttore per copia genera il problema *dell'aliasing dei puntatori.*
• Viene effettuata una copia membro a membro di tutti gli attributi della classe (i puntatori `Node* first` e `Node* last`), ma non vengono creati nuovi nodi.



Liste: inserimento

Lista non vuota:



COSTRUTTORE PER COPIA - ArrayStack

Definiamo pertanto un costruttore per copia adeguato per la classe `ArrayStack`:

```
ArrayStack::ArrayStack(const ArrayStack& o)
{
  cout<<"Copy constructor:ArrayStack(const ArrayStack& o)"<<endl;
  size=o.size;
  isize=o.isize;
  top=o.top;
  data=new char[size];
  for(int i=0; i<=top; i++)
       data[i]=o.data[i];
}
```

necessario per evitare il *pointer aliasing*

COSTRUTTORE PER COPIA - ArrayStack

Eseguendo nuovamente il test presentato nella slide n.5 `TestCopyConstructor()` si ottiene la seguente uscita (comportamento corretto):

```
ArrayStack() Dimensione iniziale 10
Raddoppiamento Array --- Nuova dimensione 20
Raddoppiamento Array --- Nuova dimensione 40
Copy constructor: ArrayStack(const ArrayStack &o)
Stack s1:z
Stack s2:z
Stack s2:y
~ArrayStack()
~ArrayStack()
```

COSTRUTTORE PER COPIA - LinkedList

Definiamo pertanto un costruttore per copia adeguato per la classe `LinkedList`:

```
LinkedList::LinkedList(const LinkedList& o)
{
  cout<<"Copy constructor: LinkedList(const LinkedList& o)"<<endl;
  first=0;
  last=0;
  for(Node *i=o.first; i!=0; i=i->GetLink())
       Add(i->GetData());
}
```


COSTRUTTORE PER COPIA - LinkedList

Eseguendo nuovamente il test `TestCopyConstructor()`, si ottiene la seguente uscita (comportamento corretto):

```
Node(a)
Node(b)
Node(c)
a b c
Copy constructor: LinkedList(const LinkedList& o)
Node(a)
Node(b)
Node(c)
a b c
~Node(b)
a c
a b c
~Node(a)
~Node(b)
~Node(c)
~Node(a)
~Node(c)
```