

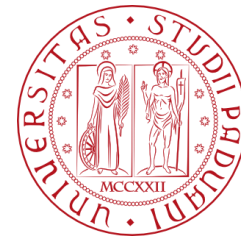
# Tutorato 2

08/11/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



Container classes are expected to implement **CRUD**-like methods to do the following:

- create an empty container (constructor);
- insert objects into the container;
- delete objects from the container;
- delete all the objects in the container (clear);
- access the objects in the container;
- access the number of objects in the container (count).

Containers are sometimes implemented in conjunction with **iterators**.

- Nel nostro caso immaginiamo di creare una classe che rappresenta un insieme di telefonate in una lista (che chiameremo `bolletta`)

# Le classi container



```
1  class telefonata
2  {
3      public:
4          telefonata(const orario&, const orario&, const numero&);
5          telefonata();
6          orario Inizio() const;
7          orario Fine() const;
8          numero Numero() const;
9          bool operator==(const telefonata&);
10
11     private:
12         orario inizio, fine;
13         numero n;
14 };
```



# Le classi container



```
1  #ifndef BOLLETTA_H
2  #define BOLLETTA_H
3  #include "telefonata.h"
4
5  class bolletta{
6      public:
7          bolletta(): first(0) {} //costr. di default
8          bolletta(const bolletta&); //costruttore di copia
9          // Metodi "utility" da container
10         bool Vuota() const; //isEmpty
11         void Aggiungi_Telefonata(telefonata); //addOne
12         void Togli_Telefonata(telefonata); //removeOne
13         void Sostituisci(const telefonata& t1, const telefonata& t2);
14         telefonata Estrai_Una(); //getOne
15     };

```



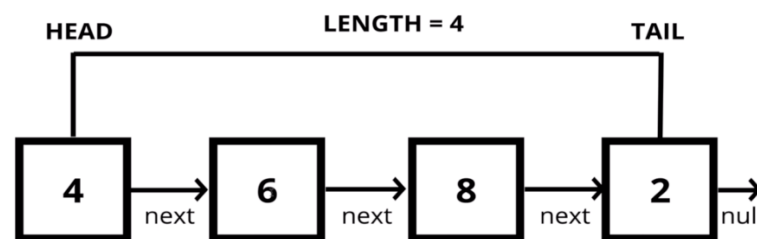
# Classi annidate (nested classes)



- Useremo una classe `nodo` per gestire i dati all'interno di bolletta

```
1  #ifndef BOLLETTA_H
2  #define BOLLETTA_H
3  #include "telefonata.h"
4
5  class bolletta{
6      private:
7          class nodo
8          {
9              public:
10                 nodo();
11                 nodo(const telefonata&, const smartp&);
12                 telefonata info;
13                 int riferimenti;
14             };
15     nodo* first; //puntiamo al primo nodo della lista
16 };
```

## Singly Linked Lists



# Le classi container



```
1
2  bool bolletta::Vuota() const {return first==0;}      //CONTROLLA SE VUOTA
3
4  void bolletta::Aggiungi_Telefonata(telefonata t)    //AGGIUNGE UNA TEL ALLA BOLLETTA
5  {
6      first=new nodo(t, first);
7  }
8
```

```
1  telefonata bolletta::Estrai_Una()                  //ESTRAE UNA TELEFONATA DALLA BOLLETTA
2  {
3      telefonata aux=first->info;
4      first=first->next;
5      return aux;
6  }
```



# Le classi container



```
1 void bolletta::Togli_Telefonata(telefonata t) //TOGLIE DALLA BOLLETTA UNA TELEFONATA T
2 {
3     smartp p=first, prec, q;
4     smartp original=first;
5     first=0;
6     while(p!=0 && !(p->info==t))
7     {
8         q=new nodo(p->info, p->next);
9         if(prec==0) first=q;
10        else prec->next=q;
11        prec=q; p=p->next;
12    }
13
14    if(p==0) {first=original;}
15
16    else if (prec==0) first=p->next;
17    else prec->next=p->next;
18 }
```



# Interferenza (o aliasing)

- Alcuni metodi creano modifiche agli oggetti di invocazione (e.g. aggiunta/rimozione telefonata)

```
1  int main(){
2      //Costruzione oggetti
3      bolletta b1;
4
5      telefonata t1(orario(9,23,12), orario(10,4,53), 2121212);
6      telefonata t2(orario(10,23,12), orario(11,4,53), 3131313);
7
8      b1.Aggiungi_Telefonata(t1);
9      b1.Aggiungi_Telefonata(t2);
10     cout << b1; //stampa della lista di telefonate
11
12     bolletta b2;
13     b2 = b1; // assegnazione
14
15     b2.Togli_Telefonata(t1); // rimozione di una telefonata
16     cout << b1 << b2;
17 }
```



# Interferenza (o aliasing)



```
1 //OUTPUT
2
3 TELEFONATE IN BOLLETTA:
4 1) INIZIO 9:23:12 FINE 10:04:53 NUMERO 2121212
5 2) INIZIO 10:23:12 FINE 11:04:53 NUMERO 3131313
6
7 //dopo toglì_telefonata
8 TELEFONATE IN BOLLETTA: //B1
9 1) INIZIO 9:23:12 FINE 10:04:53 NUMERO 2121212
10
11 TELEFONATE IN BOLLETTA: //B2
12 2) INIZIO 10:23:12 FINE 11:04:53 NUMERO 3131313
```

- L'assegnazione fa in modo entrambi puntino alla stessa area di memoria (posizione iniziale)
- La cancellazione deve essere usata con attenzione, perché potremmo puntare a zone non definite in memoria



# Interferenza (o aliasing)



- Assegnazione  $b2 = b1$  – stessa area di memoria
- `b2.Togli_Telefonata(t1)` – stesso puntatore first
- Se rimuovessi l'altra telefonata (in questo momento nulla perché copio il valore e non i campi dentro), potremmo avere situazioni *di memoria indefinita*
- Di fatto, copio i campi puntatore, ma non gli oggetti a cui puntano (perché non creo nuove istanze dell'oggetto)
- **Aliasing:** riferimenti di variabili che puntano alla stessa zona di memoria = quanto modificato un dato, l'altro involontariamente viene modificato



# Shallow copy vs deep copy



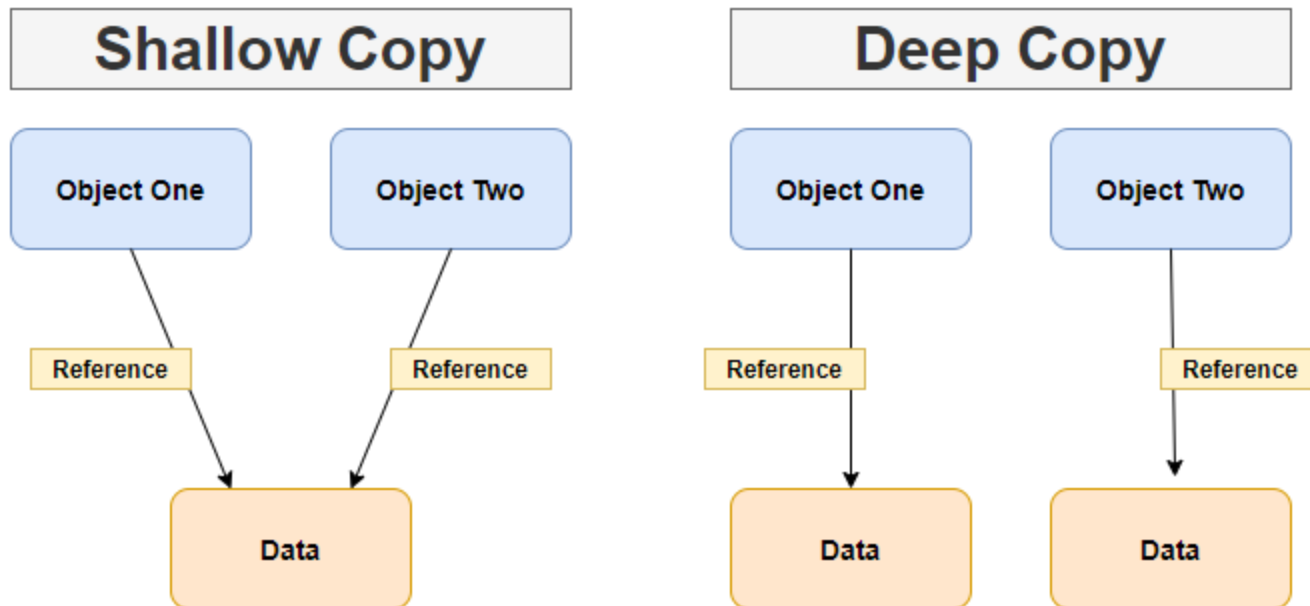
- **Shallow (superficiale)**
  - Crea un nuovo oggetto, ma non crea copie degli oggetti contenuti nell'oggetto originale. Al contrario, copia i riferimenti a tali oggetti
- **Deep (Profonda)**
  - Crea un oggetto completamente nuovo con copie di tutti gli oggetti contenuti nell'oggetto originale.
  - Questo include la copia non solo della struttura di primo livello, ma anche di tutti gli oggetti annidati.
  - Le copie profonde assicurano che il nuovo oggetto sia completamente indipendente dall'oggetto originale.



# Shallow copy vs deep copy



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



# Shallow copy vs deep copy

- Conseguenze: definiremo delle operazioni nelle classe profonde (per copiare tutti i campi e i sottocampi, essendo così sicuri di prendere tutto)
- Nel nostro contesto: nella classe `bolletta`, definiamo dei metodi per copiare e distruggere tutte le bollette e tutte le telefonate al loro interno

```
1 class bolletta{
2     private:
3
4     static nodo* copia(nodo*);
5     static void distruggi(nodo*);
6 }
7
```

```
1 bolletta::nodo* bolletta::copia(nodo* p){
2     if(p==0) return 0;
3     else return new nodo(p->info, copia(p->next));
4 }
5
6 bolletta::nodo* bolletta::distruggi(nodo* p){
7     if(p==0) return 0;
8     else{
9         distruggi(p->next);
10        delete p;
11    }
12 }
```

## Conseguenze

- Quando assegniamo la memoria, non deallochiamo i dati precedenti
- Rischio di aliasing (cioè, di puntare alla stessa area di memoria) quando assegniamo

## Risoluzione (struttura fissa):

- Controllo di non puntare alla memoria precedente (1)
- Pulisco lo heap (cioè, lo spazio puntato precedente) (2)
- Assegno tutti i campi (creo oggetto per salvare tutti i valori) (3)
- Ritorno il puntatore al contesto (4)

```
1 bolletta& bolletta::operator=(const bolletta& b)    //OPERATORE =
2 {
3     if(this!=&b){
4         distruggi(first); //distruggo tutti i nodi
5         first = copia(b.first); //copio tutti i nodi
6     }
7     return *this;
8 }
```

Operazione profonda che copia tutti i campi e i sottocampi puntati (non crea nuovi oggetti, salva riferimenti e tutti i loro valori)

Risoluzione (struttura fissa):

- Punto al primo elemento del container (1)
- Definisco un'operazione di copia di tutti i campi (2)

```
1 bolletta::bolletta(const bolletta& b) : first(copia(b.first)) {};
```

Per mantenere uno stato consistente dopo `b2.Togli_Telefonata(t1)`

Possiamo incapsulare in una classe il puntatore nodo, costruttore di copia e distruttore in una classe che «conta quanti riferimenti facciamo ad un dato».

Questo dà luogo ai cosiddetti *smart pointer*.

Esempio utilizzo: `std::shared_ptr`


- Conteggio dei riferimenti al dato a cui punta.
- Quando il conteggio dei riferimenti raggiunge zero (cioè nessun oggetto sta più puntando al dato), la memoria associata viene automaticamente liberata.

(Lo accenniamo e basta per far vedere «come fare le cose meglio»)



# Esercizio 1: Cosa Stampa

```
1  class S{
2      public:
3          string s;
4          S(string t): s(t) {}
5  };
6
7  class N{
8      private:
9          S x;
10     public:
11         N* next;
12         N(S t, N* p): x(t), next(p)
13         {cout << "N2 ";}
14
15         ~N() {
16             if(next)
17                 delete next;
18             cout << x.s + "~N ";
19         }
20     };
```



```
1  class C{
2      N* pointer;
3      public:
4          C(): pointer(0) {}
5          ~C() {delete pointer; cout << "~C ";}
6          void F(string t1, string t2 = "pippo"){
7              pointer = new N(S(t1), pointer);
8              pointer = new N(t2, pointer);
9          }
10 }
11
12 int main(){
13     C* p = new C; cout << "UNO\n";
14     p->F("pluto, paperino");
15     p->F("topolino"); cout << "DUE\n";
16     delete p; cout << "TRE\n";
17 }
```

# Esercizio 1: Cosa Stampa



```
1  /*
2  NESSUNA STAMPA UNO
3  N2 N2 N2 N2 DUE
4  pluto~N paperino~N topolino~N ~N pippo~N ~N ~C TRE
5  NESSUNA STAMPA
6  */
```



## Valore:

- Viene creato un duplicato (copia) del valore originale (uso memoria)
- La funzione lavora con una copia dei dati, e qualsiasi modifica effettuata all'interno della funzione non influisce sulla variabile o sull'oggetto originale.
- Aka = Si condivisione di memoria, modifiche solo alla variabile nella funzione, memoria non deallocata

## Riferimento:


- Viene passato un riferimento o un puntatore all'oggetto originale (non uso memoria).
- Questo significa che la funzione lavora direttamente con l'oggetto originale, e le modifiche all'interno della funzione si riflettono nell'oggetto originale.
- Aka = No condivisione di memoria, modifiche a tutti gli oggetti puntati, memoria non condivisa (meno dispendioso)

**Variabili di classe automatica:** definite dentro una funzione, deallocate al termine del blocco del programma

**Variabili di classe statica:** allocate all'inizio dell'esecuzione del programma, deallocate al termine

**Variabili dinamiche:** sempre allocate nello heap, deallocata esplicitamente con l'operatore `delete` (garbage collection)

1. Usiamo lo stesso ordine di dichiarazione dei campi dati
2. Costruiamo i campi allocando uno spazio in memoria per ogni variabile per i tipi non classe
3. Per ogni campo di tipo classe, chiamiamo costruttore default
4. Eseguiamo il corpo del costruttore



```
1 telefonata(const orario&, const orario&, const numero&);
```

- Evitano gli sprechi di memoria e rilasciano la memoria occupata
- Come per i costruttori, di default è disponibile il distruttore standard (esempio qui sotto)



```
1 ~Bolletta();
```

- Vogliamo eseguire una distruzione profonda (aka, tutta la memoria allocata dall'oggetto, compresi puntatori e riferimenti, quindi anche tutte le variabili dentro)

- **Oggetti statici:** al termine del main (per ultimi)
- **Oggetti di classe automatica:** alla fine del blocco di definizione
- **Oggetti dinamici:** chiamando `delete`

Seguono l'ordine inverso rispetto alla costruzione dei dati.

Ordine:

1. variabili locali all'uscita di una funzione
2. oggetto anonimo ritornato per valore (aka, valore passato a una funzione o `return`)
3. parametri passati per valore

# Distruttori: regole

1. Usiamo lo stesso ordine di dichiarazione dei campi dati
2. Eseguiamo il corpo del distruttore
3. Richiamo i distruttori nell'ordine dei campi dati

```
1 //Ingenuo: cancelliamo subito senza verificare se il puntatore esiste
2 bolletta::~~bolletta(){
3     distruggi(first);
4 }
5
6 //Migliore: verifichiamo se il puntatore esiste e poi cancelliamo
7 bolletta::~~bolletta(){
8     if(first) {delete first;}
9 }
```

*Regola del tre (rule of three):* distruttore, costruttore di copia, assegnazione



# Esercizio 2: Cosa Stampa

```
1 // Cosa stampa?
2
3 #include <iostream>
4 using namespace std;
5
6 class D {
7     private:
8         int z;
9     public:
10         D(int x=0): z(x) { cout << "D01 "; }
11         D(const D& a): z(a.z) { cout << "Dc "; }
12 };
13
14 class C {
15     private:
16         D d;
17     public:
18         C() : d(D(5)) { cout << "C0 ";}
19         C(int a) : d(5) { cout << "C1 ";}
20         C(const C& c) : d(c.d) { cout << "Cc ";}
21 };
```

```
1 int main() {
2     C c1;
3     cout << "UNO" << endl;
4     C c2(3);
5     cout << "DUE" << endl;
6     C c3(c2);
7     cout << "TRE" << endl;
8
9     return 0;
10 }
```

# Esercizio 2: Soluzione



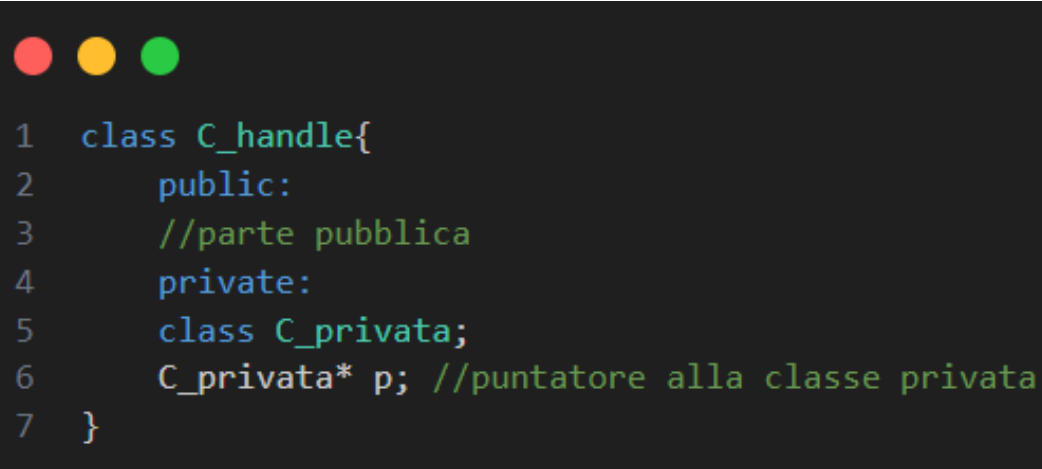
```
1  int main() {  
2      C c1;  
3      cout << "UNO" << endl; //D01 Dc C0 UNO  
4      C c2(3);  
5      cout << "DUE" << endl; // D01 C1 DUE  
6      C c3(c2);  
7      cout << "TRE" << endl; // Dc Cc TRE  
8  
9      return 0;  
10 }
```



# Gestione parte privata della classe

Talvolta, potrebbe essere desiderabile nascondere la parte privata di una classe in modo che l'utente finale non abbia accesso diretto ad essa.

Puoi definire una classe di gestione (handle) che conterrà la parte pubblica della classe.



```
1  class C_handle{
2      public:
3          //parte pubblica
4      private:
5          class C_privata;
6          C_privata* p; //puntatore alla classe privata
7  }
```

# Dichiarazione incompleta della classe

Serve a fornire un'informazione di base sul nome della classe e a consentire l'utilizzo di puntatori o riferimenti a oggetti di quella classe prima che la sua definizione completa sia disponibile.

```
1  class D; //dichiarazione incompleta
2
3  class C{
4      D* p;
5      D* m();
6      void n (... , D*, ...);
7      D& f();
8  };
9
10 //Serve a dichiarare una classe che ha un puntatore ad un'altra classe
11 //che non è ancora stata definita. In questo modo il compilatore sa che
12 //esiste una classe D, ma non sa nulla di più.
```

Piuttosto che definire una classe incompleta o un puntatore specifico, possiamo usare la keyword `friend` per dichiarare l'accesso alla parte privata o protetta della classe.

Questo ci serve per accedere a tutti gli elementi della collezione nel nostro caso

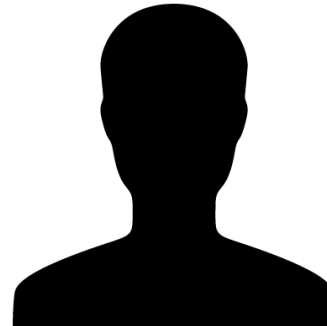
```
1 class contenitore{
2     class nodo{
3         int info;
4         nodo* next;
5         nodo(int a=0, nodo* b=0){info=a; next=b;}
6     };
7     nodo* first;
8     public:
9     class iteratore{
10         friend class contenitore; //necessaria per accedere ad iteratore
11         ...
12     };
13     contenitore(){first=0;}
14     void push(int a){first=new nodo(a,first);}
15
16 };
```

# Esempio uso classi iteratore

```
1 public:
2     class iteratore
3     {
4         friend class bolletta;
5     private:
6         bolletta:nodo* punt;
7     public:
8         bool operator==(iteratore) const;
9         bool operator!=(iteratore) const;
10        iteratore& operator++();
11        iteratore operator++(int);
12        telefonata& operator*(iteratore) const;
13    };
14    bolletta& operator=(const bolletta&);
15    iteratore begin() const;
16    iteratore end() const;
```

```
1 telefonata* operator->() const {return &(punt->info)}
2 telefonata& operator*() const {return punt->info;}
```

# Esercizio 3: Modellazione



```
1  /*Definire una classe 'Persona' i cui oggetti rappresentano anagraficamente un
2  personaggio storico caratterizzato da nome, anno dI nascita e anno di morte.
3  Includere opportuni costruttori, metodi di accesso ai campi e l'overloading
4  dell'operatore di output come funzione esterna. Separare interfaccia ed
5  implementazione della classe.
6  Si definisca inoltre un esempio al metodo 'main()' che usa tutti i metodi della
7  classe e l'operatore di output. */
8
```