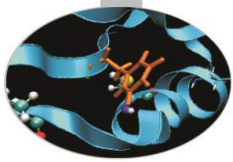
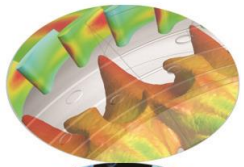
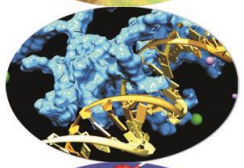
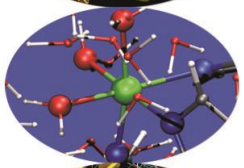
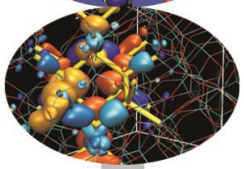
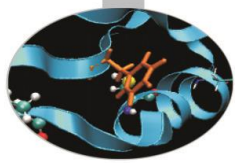


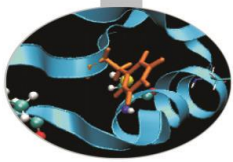
Le Classi II Parte

Indice



- Il qualificatore const
- Le funzioni friend
- Il puntatore this
- Gli operatori new e delete
- I membri static
- La composizione di classi



Le classi ed il qualificatore const

- Il qualificatore ***const*** serve per evitare che un *oggetto* o un *dato membro* di una classe venga modificato all'interno del programma. Una *funzione* dichiarata come const, invece, *non* può modificare oggetti e dati membro di una classe.
- *Funzioni membro costanti*: devono contenere il qualificatore const sia nel prototipo

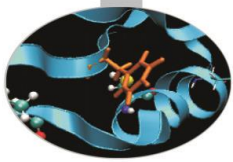
  tipo_restituito nome_funzione (lista_argomenti) ***const****;*

che nella definizione:

tipo_restituito nome_funzione (lista_argomenti) ***const****{*
corpo della funzione
}

- Gli *oggetti costanti* possono chiamare **solo** *funzioni membro costanti*, mentre gli *oggetti non costanti* possono invocare *ogni tipo di funzione*. Questo spiega perché sia buona abitudine dichiarare const tutte le funzioni membro che *non* modificano gli oggetti.
- L'unica funzione membro *non costante* che può essere chiamata da un oggetto costante è il costruttore.

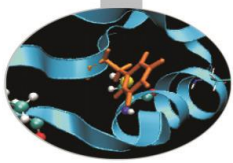
Le classi ed il qualificatore const



esempio: relazioni fra oggetti costanti e non, e funzioni membro costanti e non

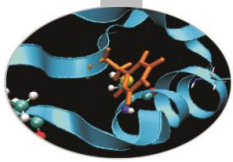
```
// header file time.h, contiene la dichiarazione della classe Time
#ifndef TIME_H
#define TIME_H
class Time{
private:
   int hour, minute, second;
public:
    Time();
    Time(int, int, int);
    int getHour() const;
    int getMinute() const;
    int getSecond() const;  // dichiariamo costanti solo le funzioni di tipo get
    void printTime();
};#endif
```

Le classi ed il qualificatore const



```
// time_fun.cc, contiene le definizioni delle funzioni
 #include<iostream.h>
 #include"time.h"
Time::Time(){ hour=0;  minute=0;   second=0;}
Time::Time(int hr, int min, int sec){
  hour=(hr < 24 && hr >=0) ? hr : 0;
  minute=(min < 60 && min >=0) ? min  : 0;
  second=(sec < 60 && sec >=0) ? sec : 0;}
int Time::getHour() const { return hour;}
int Time::getMinute() const { return minute;}
int Time::getSecond() const { return second;}
void Time::printTime() {
   cout << "Time is " << hour << ":" << minute <<  ":" << second << endl;}
```

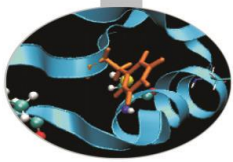
Le classi ed il qualificatore const



```
// time_prg.cc, contiene il programma vero e proprio
#include<iostream.h>
#include"time.h"
int main(){
Time sveglia(6,45,30);
const Time video(20,47,0);      // oggetto costante
```

//	oggetto	funzione membro	risultato
sveglia.printTime(); //	non costante	non costante	ok
video.printTime(); //	costante	non costante	error
cout << sveglia.getHour(); //	non costante	costante	ok
cout << video.getMinute();//	costante	costante	ok

```
return 0;
}
```



Le classi ed il qualificatore const

- *Dati membro costanti*: vengono dichiarati facendo precedere il qualificatore const al tipo di variabile cui appartengono:

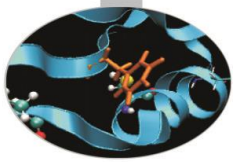
const *nome_tipo nome_variabile*;

devono, inoltre, essere inizializzati attraverso un esplicito *inizializzatore di membro*, introdotto dal segno **:** quando è chiamato il costruttore della classe.

- esempio: un valore iniziale (somma) è incremento di un fattore costante (incremento)

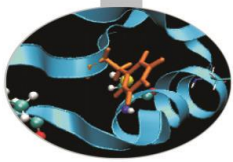
```
// header file incremento.h
#ifndef AGG_COST_H
#define AGG_COST_H
class Agg_cost{
   private:
      int somma;
      const int incremento;
   public:
      Agg_cost(int, int);
      void addIncremento();
      void print() const; };
#endif
```

Le classi ed il qualificatore const



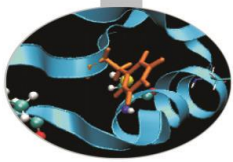
```
// file incremento_fun.cpp
#include<iostream.h>
#include"incremento.h"
Agg_cost::Agg_cost(int smm, int inc)
    :incremento(inc){ somma=smm; }
void Agg_cost::addIncremento(){
  somma+=incremento;}
void Agg_cost::print() const{
   cout << "La somma è: " << somma << "; "
        << "l' incremento è: " << incremento << endl;}
```

Le classi ed il qualificatore const



```
// file incremento_prg.cpp

#include <iostream.h>
#include “incremento.h”
int main(){
  Agg_cost valore(20, 5);
  cout << “Valori iniziali.  “;
  valore.print();
  cout << “Inizia l’incremento” << endl;
  for(int i=0; i<4; i++){
            valore.addIncremento();
            valore.print() ;}
  return 0;
}
```

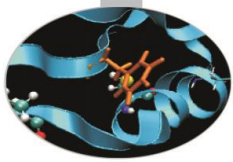
Le classi ed il qualificatore const



OUTPUT

Otteniamo il seguente output:

```
Valori iniziali. La somma è: 20; l' incremento è: 5
Inizia l'incremento
La somma è: 25; l' incremento è: 5
La somma è: 30; l' incremento è: 5
La somma è: 35; l' incremento è: 5
La somma è: 40; l' incremento è: 5
```



Le classi e le funzioni friend

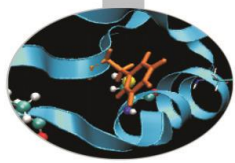
- Una funzione si dice ***friend*** di una classe quando può accedere ai membri private della classe stessa, pur essendo definita al di fuori del suo scope.

  friend *tipo_restituito nome_funzione(lista_argomenti);*

- Benché il prototipo di una funzione friend compaia all'interno della definizione di una classe, le funzioni friend *non* devono essere considerate funzioni membro.

- esempio: la funzione setVariable accede direttamente al dato membro private variable

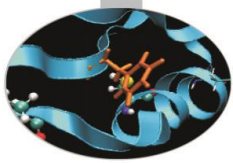
```
// header file value.h
#ifndef VALUE_H
#define VALUE_H
class Value{
   friend void setVariable(Value&, int);
   private:
     int variable;
   public:
     Value(int=0);
     void print() const;};
```



Le classi e le funzioni friend

```
// file value_prg.cpp

#include<iostream.h>
#include”value.h”
Value::Value( int vv) { variable = vv; }
void Value::print() const { cout << “Variable is: “ << variable << endl;}
void setVariable(Value &vl, int var){
      vl.variable = var; }    // la funzione friend accede direttamente a variable
int main(){
  Value attemp;
  cout << “Before calling setVariable” << endl;
  attemp.print();
```

Le classi e le funzioni friend

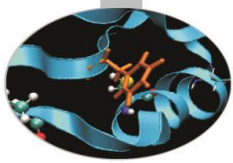


```
setVariable(attemp, 32);
   cout << “After calling setVariable” << endl;
   attemp.print();
   return 0;}
```

OUTPUT

Come output otteniamo:

```
Before calling setVariable
Variable is: 0
After calling setVariable
Variable is: 32
```



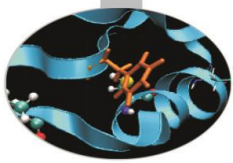
Il puntatore this

- Ogni oggetto ha la possibilità di *accedere al proprio indirizzo tramite il puntatore **this**.*
- Il puntatore this può far riferimento sia ai dati che alle funzioni membro di un oggetto. Il tipo di ***this*** dipende dal tipo dell'oggetto e dal fatto che la funzione membro in cui viene utilizzato sia dichiarata come ***const*** o meno.

Esempio:

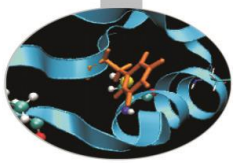
```
class A{

    public:

        void function1();   // this è di tipo A* const (puntatore costante ad un
                            //  oggetto di tipo A);

        void function2() const;  // this è di tipo const A* const (puntatore costante
                                 //   ad un oggetto di tipo A costante).

};
```

Esempio



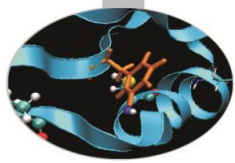
- **esempio1**: relazioni fra oggetti costanti e non, e funzioni membro costanti e non, utilizzando il puntatore this

```
// header file timet.h
#include<iostream>
#ifndef TIMET_H
#define TIMET_H
class Time{
    friend ostream& operator<<(ostream&, const Time&);
    private:
       int hour, minute, second;
    public:
       Time(int=0, int=0, int=0);
        const Time& getTime() const;
        Time& retTime();
};#endif
```



Il puntatore this

```
// file timet_fun.cc
#include<iostream.h>
#include”timet.h”
Time::Time(int hr, int min, int sec){
    hour=(hr<24 && hr>=0) ? hr : 0;
    minute=(min<60 && min>=0) ? min : 0;
    second=(sec<60 && sec>=0) ? sec : 0; }

const Time& Time::getTime() const  { return *this; }
Time& Time::retTime()  { return *this; }

ostream& operator<<(ostream& output, const Time& tm){
  output << “Time is: “ << “ “ << tm.hour << " " << tm.minute << " “
         << tm.second;
  return output;  }
```



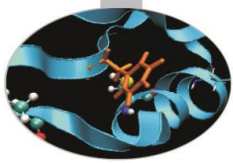
Esempio

// file timet_prg.cc

```
#include<iostrem.h>
#include"timet.h"
int main(){
        Time sveglia(6,45,30);
         const Time video(20,47,23);

         cout << sveglia.retTime() << endl;
         cout << sveglia.getTime() << endl;
         cout << video.getTime() << endl;
    /*  cout << video.retTime() << endl;   errore: un oggetto costante chiama
        una funzione noncostante  */
         return 0;  }
```

In output abbiamo:

```
Time is: 6 45 30
Time is: 6 45 30
Time is: 20 47 23
```

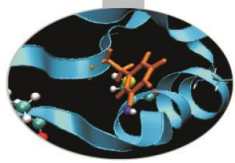


Esempio

esempio2: uso di this per far riferimento ad un dato membro

// file prova.h

```
#ifndef PROVA_H
#define PROVA_H
class Prova{
  private:
    int var;
  public:
    Prova( int = 0 );
    void print() const;
};
#endif
```

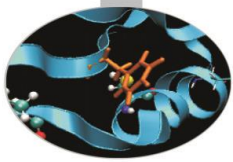


Esempio

```
// file prova_prg.cc
#include<iostream.h>
#include”prova.h”
Prova::Prova(int prv){var = prv;}
void Prova::print() const{
     cout << “output: “ << var << “ “ << this->var << “ “ << (*this).var << endl;
}

int main() {
    Prova una_var(20);
    una_var.print();
    return 0;
}
```

Come output abbiamo:

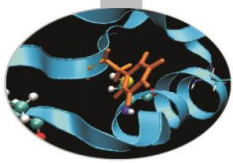
output: 20 20 20



Esempio

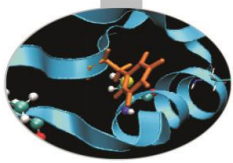
- Se le funzioni che fanno uso di this restituiscono un reference all'oggetto del quale rappresentano i metodi, allora this può essere usato per eseguire chiamate a cascata delle funzioni membro.
- **esempio 3**: utilizzo di this nelle funzioni membro per le chiamate a cascata

```
// header file time2.h
#ifndef TIME2_H
#define TIME2_H
class Time{
  private:
     int hour, minute, second;
  public:
     Time(int=0, int=0, int=0);
     Time &setTime( int, int, int );
     Time &setHour( int );
     Time &setMinute( int );
     Time &setSecond( int );
     void print() const; };#endif
```



Esempio

```
// file time2_fun.cc
 #include<iostream.h>
 #include”time2.h”
 Time::Time(int hr, int min, int sec){ setTime( hr, min, sec); }
 Time& Time::setTime(int hr, int min, int sec){
     setHour( hr );
     setMinute( min );
     setSecond( sec );
     return *this;}   //  consente le chiamate a cascata
 Time& Time::setHour( int hr ){
     hour=(hr < 24 && hr >=0) ? hr : 0;
     return *this; }
  Time& Time::setMinute ( int min ){
     minute=(min < 60 && min >=0) ? min  : 0;
     return *this; }
```

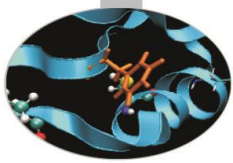


Esempio

```
Time& Time::setSecond( int sec ){
      second=(sec < 60 && sec >=0) ? sec : 0;
      return *this; }
  void Time::print() const{
      cout << “Time is: “ << hour << “:” << minute << “:” << second << endl;}
```

OUTPUT
Time is: 10:20:32
Time is: 16:36:54

```
// file time2_prg.cc
#include<iostream.h>
#include”time2.h”
int main(){
   Time t;
   t.setHour(10).setMinute(20).setSecond(32).print();  // chiamata a cascata
   t.setTime(16, 36, 54 ).print();      // chiamata a cascata
   // t.print().setTime(19, 31, 22).print()   errore nella prima chiamata a print
   return 0;
}
```



Esempio

- Nell'espressione:

t.setHour(10).setMinute(20).setSecond(32).print();

sfruttando l'associatività dell'operatore **.** da sx a dx viene valutata per prima la parte *t.setHour(10)* che restituisce, tramite **this*, un reference a t. Il resto dell'espressione è, allora, equivalente a:

t.setMinute(20).setSecond(32).print();

e così via.

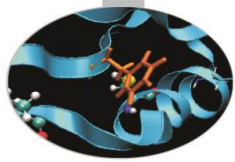
- L'espressione commentata

t.print().setTime(19, 31, 22).print();

contiene, invece, un errore di sintassi perché la funzione print() restituisce un tipo *void* dunque, dopo aver valutato *t.print()* il resto dell'espressione **non** è equivalente a:

t.setTime(19, 31, 22).print(); .

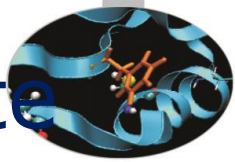
Le classi e gli operatori new e delete



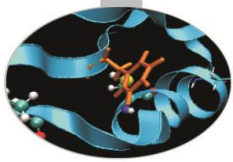
Poiché le classi non sono altro che tipi di dati definiti dagli utenti, gli operatori ***new*** e ***delete*** possono allocare dinamicamente memoria anche per gli oggetti.

```
class Name_class {… };

Name_class *objectPtr;

objectPtr = new Name_class
(parametri_di_inizializzazione);

…

delete objectPtr;
```

Le classi e gli operatori new e delete



- E' importante ricordare che in presenza dell'operatore *new* all'interno del *costruttore* di un oggetto è necessario inserire l'operatore *delete* nel corpo del *distruttore* dell'oggetto medesimo affinché sia rilasciata la memoria allocata in precedenza.
- I comandi new e delete applicati a *puntatori a oggetti* ne chiamano automaticamente il costruttore ed il distruttore.



Le classi ed i membri static

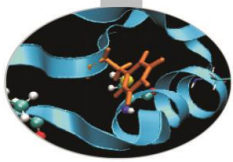
- Ciascun oggetto di una classe possiede una propria copia di tutti i dati membro della classe ma, in alcuni casi, potrebbe essere utile che *una sola copia* di un dato membro sia *condivisa* tra tutti gli oggetti della classe. Per soddisfare a questa condizione una variabile deve essere dichiarata con l'attributo ***static***.

static *nome_tipo nome_variabile;*

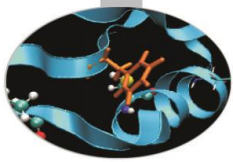
- I dati membro static hanno scope di classe, possono essere public, private o protected e vanno *inizializzati* una sola volta nello scope di file (il che significa al di fuori del costruttore), utilizzando il nome della classe cui appartengono seguito dall'operatore di risoluzione dello scope **::**

nome_tipo nome_classe **::** *nome_variabile = valore_iniziale ;*

Le classi ed i membri static



- Un dato membro *public* dichiarato static può essere acceduto direttamente da ogni oggetto della classe o attraverso *il nome della classe* seguito dall'operatore di risoluzione dello scope.
- Un dato membro *private* dichiarato static può essere acceduto tramite le funzioni public o friend della classe.



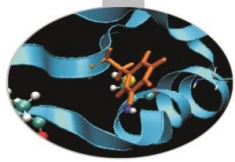
Le classi ed i membri static

- Un dato membro static esiste anche se non è stato ancora creato nessun oggetto. Per accedervi, in questo caso, se il dato membro è *public* è sufficiente utilizzare il nome della classe seguito dall'operatore di risoluzione dello scope e dal nome del dato membro:

nome_classe :: nome_variabile;

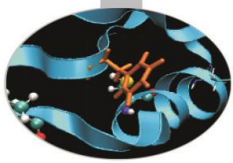
- mentre, se il dato membro è *private*, occorre definire, tra i metodi public, una *funzione membro static* che restituisca il dato membro in questione e chiamarla, poi, insieme con il nome della classe e con l'operatore di risoluzione dello scope:

nome_classe :: nome_funzione_membro_static();



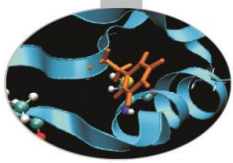
Le classi ed i membri static

- Le funzioni membro static, definite come public, possono essere chiamate, naturalmente, tramite gli oggetti della classe cui appartengono, ma è *preferibile* invocarle attraverso il nome della classe stessa seguito dall'operatore di risoluzione dello scope.
- Una funzione membro static *non* possiede un puntatore this perché dati e funzioni membro static esistono indipendentemente dagli oggetti della classe.
- Le funzioni membro static possono accedere **solo** ai dati membro static.



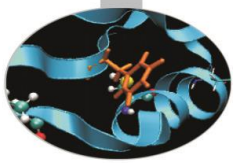
Esempio

```
// header file impiego.h
#ifndef IMPIEGO_H
#define IMPIEGO_H
class Impiegati{
    private:
      char* nome;
      char* cognome;
      static int counter;     // dato membro static, conta il numero di oggetti
istanziati
    public:
      Impiegati( const char*, const char* );
      ~Impiegati();
      const char* getNome() const;
      const char* getCognome() const;
      static int getCounter(); }; /* funzione membro static, può accedere solo a
                                  counter e restituisce il numero di oggetti istanziati  */
#endif
```



Esempio

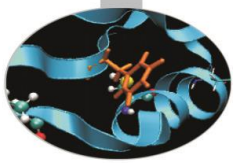
```
// file impiego_fun.cc
#include<iostream.h>
#include<string.h>
#include”impiego.h”
int Impiegati::counter=0; // inizializzazione del dato membro static
int Impiegati::getCounter() { return counter; }      // definizione della funzione
                                                     // membro static
Impiegati::Impiegati( const char *var_n, const char * var_c) {
    nome = new char [strlen( var_n )+1];    // allocazione di memoria con new
    strcpy(nome, var_n);
    cognome = new char [strlen( var_c )+1];  // allocazione di memoria con
new
    strcpy(cognome, var_c);
    ++ counter;
    cout << “E’ stato chiamato il costruttore per “
         <<  nome << “ “ << cognome << endl;
}
```



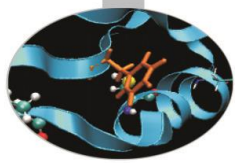
Esempio

```
Impiegati::~Impiegati() {
    cout << “E’ stato chiamato il distruttore per “
         << nome << “ “ << cognome << endl;
    delete nome;       // deallocazione della memoria con delete
    delete cognome;  // deallocazione della memoria con delete
    --counter;
  }
 const char *Impiegati::getNome() const { return nome; }
 const char *Impiegati::getCognome() const { return cognome; }
```

Esempio



```
// file impiego_prg.cc
#include<iostream.h>
#include"impiego.h"
int main(){
  cout << "Numero di impiegati (nessuna istanza): "
       << Impiegati::getCounter() << endl; // è stato usato il nome della
classe
  Impiegati *imp1_ptr = new Impiegati( "Mario", "Marchisio" );   //
costruttore
  cout << " Numero di impiegati : " << imp1_ptr->getCounter() << endl;
  Impiegati *imp2_ptr = new Impiegati( "Raffaele", "Ponzini" );  // costruttore
  cout << " #  di impiegati: " << Impiegati::getCounter() << endl;
                          // abbiamo usato ancora il nome della classe
  cout << "Impiegati: " << endl;
  cout << imp1_ptr->getNome() << " " << imp1_ptr->getCognome() << endl;
  cout << imp2_ptr->getNome() << " " << imp2_ptr->getCognome() << endl;
  delete imp1_ptr;  // viene invocato il distruttore
  cout << " #  di impiegati: " << Impiegati::getCounter() << endl;
  delete imp2_ptr; // viene invocato il distruttore
  cout << " #  di impiegati: " << Impiegati::getCounter() << endl;
  return 0; }
```



Esempio

L'output generato dal programma appena visto è il seguente:

Numero di impiegati (nessuna istanza): 0

E' stato chiamato il costruttore per Mario Marchisio

Numero di impiegati : 1

E' stato chiamato il costruttore per Raffaele Ponzini

di impiegati: 2

Impiegati:

Mario Marchisio

Raffaele Ponzini

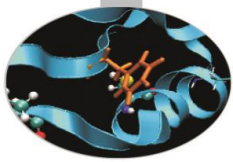
E' stato chiamato il distruttore per Mario Marchisio

di impiegati: 1

E' stato chiamato il distruttore per Raffaele Ponzini

di impiegati: 0

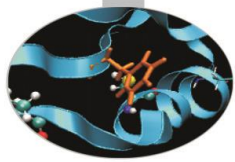
Le classi: il concetto di composizione



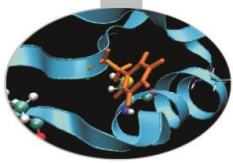
- La *composizione* è una funzionalità del C++ in virtù della quale una classe può contenere, come membri, oggetti di altre classi.
- esempio: definiamo una nuova classe Data e con questa ampliamo la classe Impiegati.

```
class Data{
    private:
      int giorno, mese, anno;
    public:
      Data(int=1, int=1, int=1900);
      ~Data();
      void print() const;
};
```

Esempio



```
class Impiegati{
    private:
      char* nome;
      char* cognome;
      Data  nascita;
      Data  assunzione;
    public:
       Impiegati( const char*, const char*, int, int, int, int, int, int );
      ~Impiegati();
      const char* getNome() const;
      const char* getCognome() const;
};
```



Esempio

- Il costruttore della classe Impiegati viene riscritto nel seguente modo:

```
Impiegati(const char* nome, const char* cognome,
          int n_giorno, int n_mese, int n_anno,
          int a_giorno, int a_mese, int a_anno)
        : nascita(int n_giorno, int n_mese, int n_anno),
          assunzione(int a_giorno, int a_mese, int a_anno)
  {   … }
```

- ovvero utilizzando gli *inizializzatori di membro* che sono separati dalla lista degli argomenti dal segno **:** ed indicano i parametri da passare ai costruttori degli *oggetti membro*. In assenza degli inizializzatori di membro vengono chiamati automaticamente i costruttori di default degli oggetti membro.
- N.B.: i costruttori sono chiamati dall'interno verso l'esterno ed i distruttori in ordine inverso. Nell' esempio, quindi, gli oggetti della classe Data sono i primi ad essere costruiti e gli ultimi ad essere distrutti.