object oriented programming

Alberto Ferrari

object oriented programming

- la ***programmazione orientata agli oggetti*** (object oriented programming) è un ***paradigma di programmazione***
- permette di raggruppare in un'unica entità (la ***classe***)
  - le strutture ***dati***
  - le ***procedure*** che operano su di esse
- si creano "***oggetti***" software dotati di ***proprietà*** (dati) e ***metodi*** (procedure) che operano sui dati dell'oggetto stesso

object oriented design

- la ***progettazione*** orientata agli oggetti ha l'obiettivo di formalizzare gli oggetti del mondo reale e di costruire con questi un mondo virtuale
- si avvale del concetto di ***classe***: gli ***oggetti*** di una determinata classe hanno le stesse ***caratteristiche***
- questa parte di mondo che viene ricostruita in modo virtuale è detta ***dominio applicativo***

il livello di astrazione

- i linguaggi di programmazione si sono evoluti in modo che i codici sorgenti potessero ***astrarsi*** sempre più dal modo in cui gli stessi, una volta compilati, sarebbero stati eseguiti
- nella OOP non ci si vuole più porre i problemi dal punto di vista del calcolatore, ma si vogliono risolvere facendo ***interagire oggetti*** del dominio applicativo come fossero oggetti del mondo reale
- l'obiettivo è di dare uno strumento al programmatore, per formalizzare soluzioni ai propri problemi, pensando come una persona e senza doversi sforzare a pensare come una macchina

programmazione imperativa vs programmazione ad oggetti

- linguaggi procedurali
  - nei linguaggi procedurali (C, Fortran, Pascal) la programmazione è orientata all'***azione***
  - l'unità di programmazione è la ***funzione***
  - ***metodologia***: ***scomposizione funzionale***
- linguaggi a oggetti
  - nel linguaggi ad oggetti (C++, Java) la programmazione è orientata all'***oggetto***
  - l'unità di programmazione è la ***classe***
  - ***metodologia: object oriented design***

il processo di astrazione: le classi

- per ***popolare*** il dominio applicativo utilizzato dall'applicazione è necessario creare gli ***oggetti***, e per fare questo è necessario definire le ***classi***
- una classe è lo strumento con cui si ***identifica*** e si ***crea*** un oggetto

classi e tipi di dato

- una ***classe*** è a tutti gli effetti un ***tipo di dato*** (come gli interi e le stringhe e ogni altro tipo già definito)
- un tipo di dato consiste di
  - un insieme di ***valori***
  - un insieme di ***operazioni***
- nella programmazione orientata agli oggetti, è quindi possibile sia ***utilizzare tipi*** di dato ***esistenti***, sia ***definirne*** di ***nuovi*** tramite le classi
- si definisce tipo di dato astratto (ADT – ***abstract data type***)
  - se il suo utilizzo è ***indipendente*** dall'***implementazione*** dei valori e delle operazioni
- i tipi di dati ***predefiniti*** sono ***ADT***
- le classi ***devono essere ADT***

UML (Unified Modeling Language)

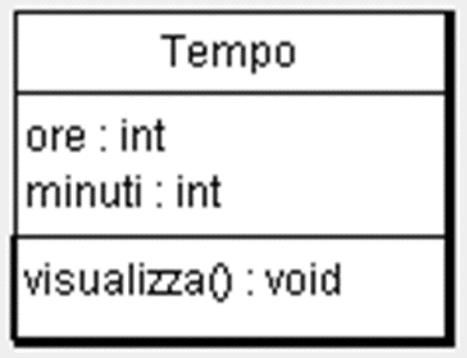
- UML ("linguaggio di modellazione unificato") è un linguaggio di modellazione e specifica basato sul paradigma object-oriented
- il nucleo del linguaggio fu definito nel ***1996*** … il linguaggio nacque con l'intento di unificare approcci precedenti (dovuti ai tre padri di UML e altri), raccogliendo le best practices nel settore e definendo così uno standard industriale unificato
- … gran parte della letteratura di settore usa UML per ***descrivere*** soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico.

http://www.uml.org/

wikipedia

UML (diagramma delle classi)

- la prima sezione contiene il ***nome della classe***
- la seconda sezione definisce i suoi ***attributi***
  - (in C++ ***variabili membro***)
- nella terza sezione sono definiti i ***metodi***, le ***operazioni*** che si possono compiere sull'oggetto di quel tipo
  - (in C++ ***funzioni membro***)



esempio C++

```
#include <iostream>
using namespace std;

class Tempo {
public:
    int ore;
    int minuti;
    void visualizza() {
     cout<<ore<<":"<<minuti<<endl;
    }
private:
};
```

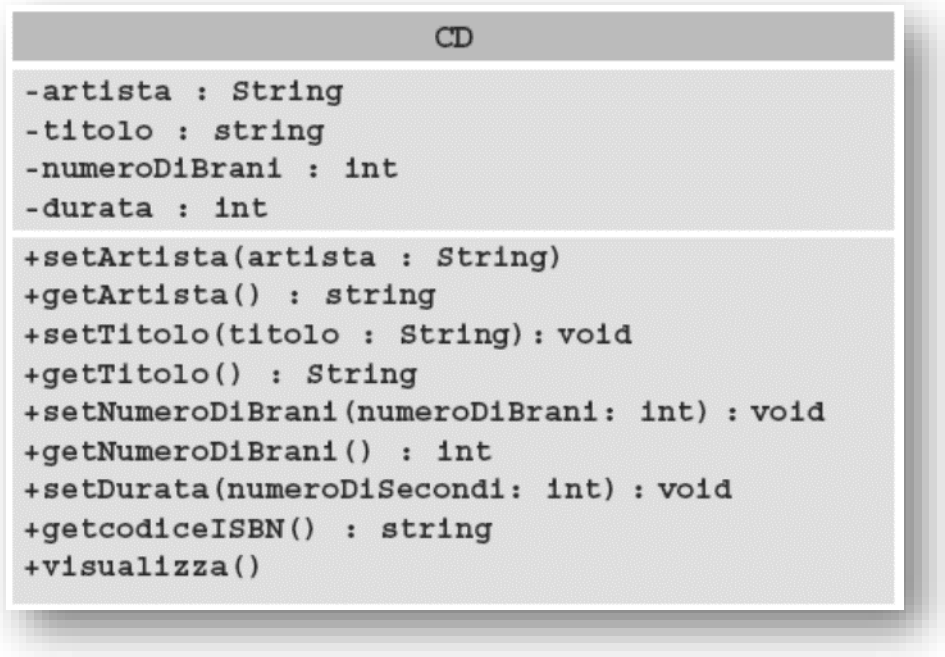
```
int main() {
    Tempo t;
    t.ore=9;
    t.minuti=30;
    t.visualizza();
    return 0;
}
```

gli oggetti

- gli oggetti sono le ***entità*** di un programma che ***interagiscono*** tra loro per raggiungere un ***obiettivo***
- gli oggetti vengono ***creati*** in fase di ***esecuzione*** ed ognuno di essi fa parte di una categoria (di una ***classe***)
- ogni classe può creare ***più oggetti***, ognuno dei quali pur essendo dello ***stesso tipo*** è ***distinto*** dagli altri
- un ***oggetto*** è l'***istanza*** di una ***classe***

un esempio di classe

- se vogliamo catalogare i ***cd musicali*** in nostro possesso, abbiamo bisogno di implementare un programma nel cui ***dominio applicativo*** è presente la ***classe CD***
- i ***metodi*** (funzioni membro) della classe CD servono per impostare e recuperare i valori degli attributi (variabili membro)



stato di un oggetto e diagramma degli oggetti

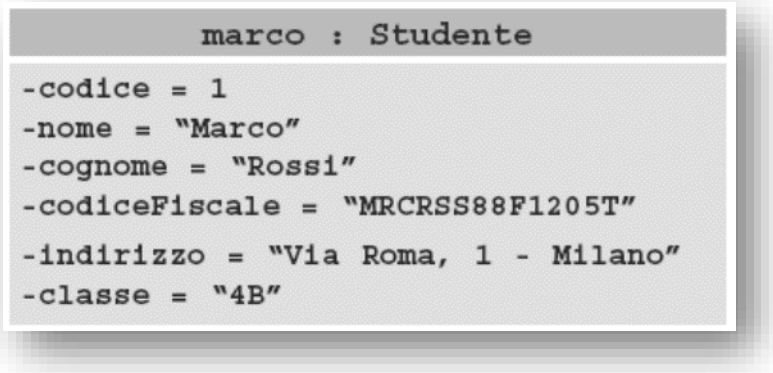
- i diagrammi che rappresentano gli oggetti
  - (***Object Diagram*** in UML)
- mettono in evidenza i ***valori*** che assumono gli attributi
- si definisce ***stato*** di un oggetto l'insieme dei ***valori degli attributi***
- lo stato dell'oggetto può variare in funzione del tempo

creazione di un oggetto

- per creare un oggetto si effettua un'***istanziazione*** di una classe
- in questa fase viene riservato uno ***spazio di memoria*** per conservare i valori degli ***attributi*** dell'oggetto che si sta creando
  - (mantenere memorizzato parte lo stato dell'oggetto)
- i vari linguaggi utilizzano diversi costrutti di programmazione per creare un oggetto

variabili membro

- le ***variabili membro*** sono quelle posseduti da un oggetto, sono chiamate anche ***attributi*** dell'oggetto
- l'attributo di un oggetto è una variabile che ne descrive una ***caratteristica*** o proprietà



le azioni degli oggetti

- una ***funzione membro*** (***metodo***) è un'azione che l'oggetto può eseguire
- la ***dichiarazione*** di una funzione è composta da:
  - ***nome*** del metodo
  - ***tipo*** di dato da ***ritornare***
  - ***tipo*** e nome dei ***parametri*** di ingresso
- l'insieme formato dal tipo del metodo, dal nome e dal tipo dei parametri è detto ***signature*** (***firma*** del metodo)
- una funzione membro, per essere ***utilizzata***, ha bisogno della creazione di un ***oggetto*** della classe a cui appartiene ***su cui essere invocata***

metodo costruttore

- il ***costruttore*** è un metodo particolare che viene ***invocato*** alla ***creazione*** dell'oggetto e che contiene tutte le ***istruzioni*** da eseguire per la sua ***inizializzazione***
- deve avere lo ***stesso nome*** della ***classe*** e non può ritornare un valore
- deve stare nella sezione pubblica della classe
- spesso si hanno più costruttori (overloading)
- un costruttore ***senza argomenti*** è detto costruttore di ***default***
  - se non definiamo nessun costruttore viene creato un costruttore di default
  - se definiamo almeno un costruttore il costruttore di default non viene creato
  - è bene includere sempre il costruttore di default

- quando si ***dichiara*** una variabile di tipo classe e si vuole invocare il ***costruttore senza argomenti***, non si usano le parentesi
  - esempio: `Data oggi;`
- il costruttore può essere chiamato esplicitamente per modificare le variabili membro di un oggetto
  - crea un oggetto anonimo e lo inizializza con i valori degli argomenti
  - l'oggetto anonimo può essere assegnato a una variabile del tipo classe
  - esempio:
```
Data d;
d = new Data(27,4);
```

modificatori di visibilità

- ***public***
  - consente a ***qualunque classe o oggetto*** di qualsiasi tipo di avere ***accesso*** all'attributo o al metodo a cui è applicato
- ***protected***
  - consente l'accesso solo alle classi e agli oggetti il cui tipo è una ***sottoclasse*** di quella in cui è utilizzato
    - *le sottoclassi saranno trattate in successive lezioni*
- ***private***
  - consente l'accesso ***solo*** agli oggetti della ***classe stessa*** in cui è definito

incpsulamento

- l'incapsulamento (***information hiding***) è un concetto fondamentale dell'ingegneria del software
- questo principio prevede che si possa ***accedere*** alle informazioni di un oggetto ***unicamente attraverso i suoi metodi***
- la tecnica di programmazione che consente di applicare l'incapsulamento si avvale dei modificatori di visibilità per ***nascondere gli attributi*** di un oggetto al mondo esterno
- mettere in atto questa tecnica significa non avere ***mai*** attributi di un oggetto di tipo ***public***, salvo eccezioni particolari per costanti o attributi di classe da gestire in base al caso specifico

accesso agli attributi

- è buona norma rendere ***private*** tutte le ***variabili membro*** e ***pubbliche*** solo le ***funzioni*** membro ***necessarie*** (quelle che espongono le funzionalità della classe)
- per accedere dall'esterno agli attributi, si inseriscono ***metodi public*** che possono essere chiamati da chiunque per ***impostare*** o ***richiedere*** il valore dell'attributo
- i metodi hanno di solito un nome particolare:
  - ***set*** (seguito dal nome dell'attributo) per ***modificarne*** il valore
    - metodi ***setter*** (metodi ***mutator***)
  - ***get*** (seguito dal nome dell'attributo) per ***recuperare*** il valore
    - metodi ***getter*** (metodi ***accessor***)

incapsulamento: perché?

- potrebbe sembrare che non vi sia alcuna differenza rispetto ad accedere direttamente agli attributi
- ***sembra*** che questa tecnica serva solo a rendere più ***complessa*** la loro gestione
- le ***motivazioni*** sono:
  - un maggiore ***controllo*** sulle operazioni effettuate sugli attributi, ***limitando l'utilizzo improprio*** che se ne può fare e guadagnando così in ***sicurezza***
  - la possibilità di ***nascondere*** il modo in cui i dati sono memorizzati negli attributi

interazione fra oggetti

- per comunicare, gli oggetti possono utilizzare i metodi, ***scambiandosi messaggi*** l'uno con l'altro
- quando un oggetto invia un ***messaggio*** a un altro oggetto, quest'ultimo reagisce eseguendo il ***metodo*** opportuno
- l'invocazione dei metodi può richiedere ***parametri*** di input di ***qualsiasi tipo***, compresi quindi ***oggetti*** del nostro dominio applicativo
- un oggetto potrà quindi essere in grado di passarne un altro attraverso un metodo, o addirittura potrà passare se stesso
- un messaggio ha la seguente sintassi:

```
<nomeOggetto>.<nomeMetodo>(<paramteri>)
```

interfaccia di una classe

- ***interfaccia*** di una classe:
  - ***dichiarazioni*** delle ***funzioni membro pubbliche***
  - ***commenti***
- ***implementazione*** di una classe:
  - ***variabili membro*** e ***dichiarazioni*** delle ***funzioni membro private***
  - ***definizioni*** delle ***funzioni membro***
- l'***interfaccia*** viene generalmente definita in un ***file header***
- il file header sarà ***incluso*** da tutti i file che vogliono fare uso della classe
  - (direttiva **#include**)
- chi vende librerie software, fornisce ai clienti i soli file header e il codice oggetto

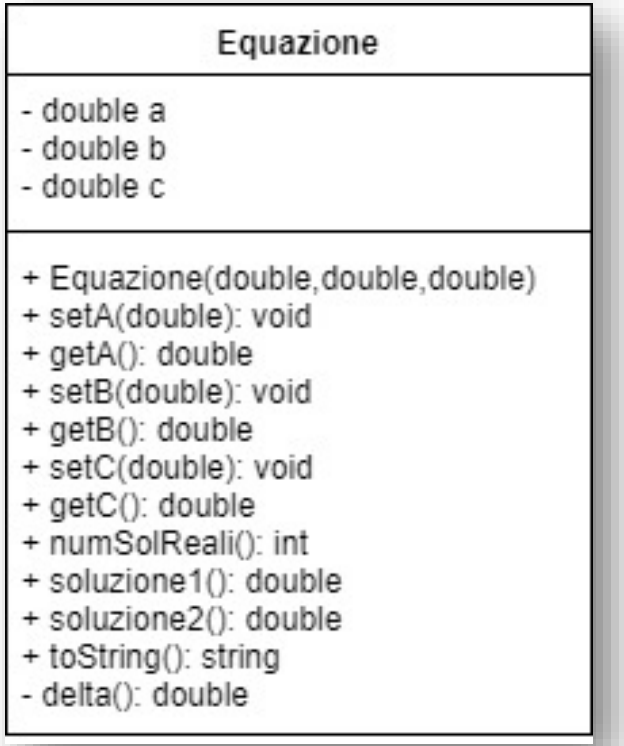
utilizzo delle classi

- deve essere possibile ***utilizzare*** una classe conoscendone ***solo*** l'interfaccia
  - ***vantaggio***: è possibile ***cambiare l'implementazione senza*** dover ***cambiare*** qualsiasi altro ***codice*** che usi la classe
- ***filosofia***: descrivere il problema in termini di oggetti che interagiscono,
- piuttosto che algoritmi che operano su dati (anche algoritmi e dati possono cambiare...)

un esempio

- si vuole realizzare una classe che permetta di gestire e risolvere ***equazioni di secondo grado***
- in una equazione individuiamo tre ***attributi***: ***a, b, c*** che rappresentano i ***coefficienti*** di x^2, di x ed il termine noto
- l'equazione $3x^2 - 2x + 1 = 0$ avrà come attributi i valori 3, -2 e 1
- definiamo un insieme di ***metodi*** che ci permetta di:
  - modificare i valori dei coefficienti
  - ottenere i valori dei coefficienti
  - conoscere il tipo di equazione
  - ottenere la prima soluzione
  - ottenere la seconda soluzione

diagramma UML della classe



esercizio

- implementare la classe Equazione
- istanziare due equazioni:

  $5x^2-3x+2=0$

  $2x^2-4=0$

eq1 : Equazione
a : double = 5
b : double = -3
b : double = 2

eq2 : Equazione
a : double = 2
b : double = 0
b : double = -4

soluzione esercizio

```
#include <iostream>
#include <cmath>
#include <limits>
#include <string>
using namespace std;
class Equazione {
   public:
     Equazione() { a=1; b=1; c=1; }
     Equazione(double c_a, double c_b, double c_c){
       a=c_a; b=c_b; c=c_c;
     }
   double getA() { return a; }
   void setA(double v) { a=v; }
   …
   //  numero di soluzioni reali
   int numSolReali()  {
     if (a==0 && b!=0) return 1;
     if (a==0 && b==0) return 0;
     if (delta()<0)    return 0;
     if (delta()==0)   return 1;
     return 2;
   }
```

```
   double soluzione1() {
      if (a==0 && b!=0) return (-c/b);
      if (delta()>=0)    return ((-b-sqrt(delta()))/a);
      return std::numeric_limits<double>::min();
   }
   double soluzione2() {
      …
   }
   string toString() {
      string s = "";
      if (a!=0) s += to_string(a) + "x^2 ";
      if (b>0) s += "+" + to_string(b) + "x ";
      if (b<0) s += to_string(b) + "x ";
      if (c>0) s += "+" + to_string(c);
      if (c<0) s += to_string(c);
      return s + " = 0";
   }
   private:
      double a;
      double b;
      double c;
      double delta() { return pow(b,2)-4*a*c; }
};
```

soluzione esercizio

```
int main() {
   Equazione e(3,4,1);
   e.setA(2);
   cout << e.toString() << endl;
   switch (e.numSolReali()) {
     case 2:
       cout << "soluzione 2 = " << e.soluzione2() << endl;
     case 1:
       cout << "soluzione 1 = " << e.soluzione1() << endl;
       break;
     case 0:
       cout << "nessuna soluzione reale";
   }
   return 0;
}
```

- **`Equazione e(3,4,1)`**
  - dichiarazione e inizializzazione (chiamata costruttore)
- **`e.setA(2)`**
  - esecuzione della funzione membro void setA(double)
- **`e.toString()`**
  - esecuzione della funzione membro string toString()
- **`e.numSolReali()`**
  - esecuzione della funzione membro int numSolReali()
- **`e.soluzione1()`** e.soluzione2()
  - esecuzione della funzione membro double soluzione1()

lista di inizializzazione

- il costruttore ha il compito di ***inizializzare*** le ***variabili membro*** della classe
- può eseguire questa operazione in due modi
  - tramite ***assegnamenti*** o chiamate a funzioni all'interno del ***corpo*** del costruttore stesso

```
Equazione(double c_a, double c_b, double c_c){
        a=c_a; b=c_b; c=c_c;
 }
```

  - tramite ***lista di inizializzazione***, una sequenza `Attributo(<Espressione>)` che indica al compilatore di memorizzare il valore dell'Espressione in Attributo

```
Equazione(double c_a, double c_b, double c_c): a(c_a), b(c_b, c(c_c){}
```

suddivisione del codice in file

- ***separazione*** tra la ***classe*** e i ***programmi*** che la usano
  - ***riuso***: parti separate facilmente riusabili (*libreria*)
  - ***compilazione selettiva***
- ***separazione*** tra ***interfaccia*** e implementazione
  - ***incapsulamento***: occultamento dei dettagli
  - ***diverse implementazioni*** di una stessa libreria

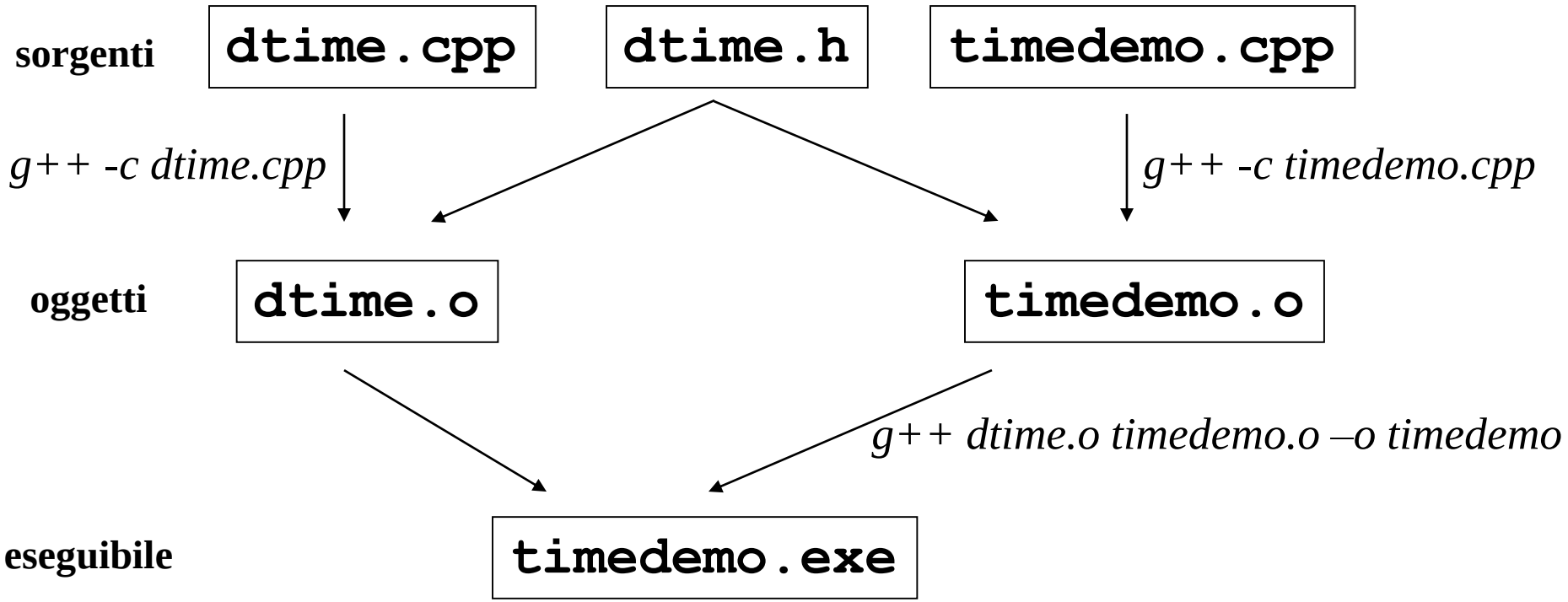
regole per incapsulamento

- rendere ***private*** tutte le ***variabili membro***
- raggruppare ***definizione*** della classe, ***dichiarazioni*** delle ***funzioni membro*** e commenti nel **file di interfaccia** (***header file***)
- raggruppare le ***definizioni*** delle ***funzioni membro*** e l'inizializzazione delle variabili *static* nel ***file di implementazione***

compilazione separata

- il file che contiene il ***programma*** che usa la classe si chiama ***file di applicazione***
- sia l'***implementazione*** che l'***applicazione*** devono ***includere*** l'***header*** file
- l'implementazione e l'applicazione vengono ***compilate separatamente***
- per ottenere l'eseguibile occorre linkare i due oggetti

compilazione separata



vantaggi della compilazione separata

- ***separando*** l'***interfaccia*** e l'***implementazione*** della classe dall'applicazione
  - posso ***riusare*** la classe in diversi programmi senza riscriverla
  - posso ***compilare*** l'***implementazione una sola volta***
- separando l'***interfaccia*** dall'***implementazione***
  - se ***cambio*** l'***implementazione non*** devo cambiare i ***programmi*** che usano la classe
  - devo solo ***ricompilare*** l'implementazione e ***rilinkare***

inclusione ripetuta

classA.cpp

classA.h

```
class A
{
  public:
  …
  private:
  …
};
```

classB.cpp

classB.h

```
#include "classA.h"
class B
{
  public:
  …
  private:
    A var;
};
```

Main.cpp

```
#include "classA.h"
#include "classB.h"

…
```

uso di `#ifndef`

- un ***header*** file può ***includere altri header*** file
- per ***evitare*** che il contenuto di un header file venga ***incluso più volte*** racchiudo il codice tra

```
#ifndef NOMEHEADER_H
#define NOMEHEADER_H
e
#endif
```

- per ***convenzione*** si usa il *nome del file in maiuscolo* e con l'*underscore* al posto del punto
- *usato in tutti gli header std (iostream, vector, string,…)*

esempio: inclusione ripetuta

classA.cpp

classA.h

```
#ifndef CLASSA_H
#define CLASSA_H
class A
{
   …
};
#endif
```

classB.cpp

classB.h

```
#ifndef CLASSB_H
#define CLASSB_H
#include "classA.h"
class B
{
   …
};
#endif
```

membri static: funzioni

- una ***funzione*** membro ***static*** accede solo ai ***membri static***
- ***non*** può accedere ai dati dell'***oggetto*** chiamante
- viene invocata usando il ***nome della classe*** e lo scope resolution operator (**::**)
- la parola chiave ***static*** va messa ***solo nella dichiarazione***

membri static: variabili

- una ***variabile*** membro ***static*** è condivisa da tutti gli oggetti di una classe
- è usata dagli oggetti della classe per ***comunicare*** e coordinarsi
- ***solo*** gli ***oggetti della classe*** possono accedervi
- va ***inizializzata*** al di fuori della definizione della classe, una sola volta

esempio: Cameriere.h

```
#ifndef CAMERIERE_H
#define CAMERIERE_H
#include<string>
using namespace std;
class Cameriere {
public:
   Cameriere(string nomeCameriere);
   static int getClienteDaServire();
   void serviCliente();
   static bool isAperto();
private:
    static int clienteDaServire;
    static int ultimoServito;
    static bool servizioAperto;
    string nome;
};
#endif
```

- variabili membro static provate:
- clienteDaServire
  - numero dell'ultimo cliente da servire
- ultimoServito
  - numero dell'ultimo cliente servito
- servizioAperto
  - true se ci sono ancora clienti da servire

- funzioni membro static pubbliche:
- getClienteDaServire()
- isAperto()
  - accesso alle variabili static

membri static: esempio

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

int Cameriere:: clienteDaServire = 0;
int Cameriere:: ultimoServito = 0;
bool Cameriere::servizioAperto = true;

Cameriere::Cameriere(string nomeC) : nome(nomeC)
{/*Intentionally empty*/}

int Cameriere::getClienteDaServire(){
    clienteDaServire++;
    return clienteDaServire;
}

bool Cameriere::isAperto(){
    return servizioAperto;
}
```

```
void Cameriere::serviCliente( ){
    if (servizioAperto &&
        ultimoServito < clienteDaServire){
        ultimoServito++;
        cout << "il cameriere " << nome
            << " sta servendo il cliente numero«
            << ultimoServito << endl;
     }

    if (ultimoServito >= clienteDaServire)
        servizioAperto = false;
}
```