



UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI
MATEMATICA E INFORMATICA

Introduzione alla OOP

Alessandro Ortis

ortis@dmf.unict.it
www.dmf.unict.it/ortis/

L'importanza dell'astrazione

Il mondo in cui viviamo è costituito da sistemi molto complessi, in cui oggetti diversi interagiscono tra loro e cambiano il loro modo di agire in funzione di quello che accade.

È difficile gestire una realtà complessa, allo stesso modo è difficile costruire sistemi complessi come ad esempio software di grandi dimensioni.

Un modo per gestire la complessità è l'**astrazione**.

L'importanza dell'astrazione

Esempi di astrazione...

L'importanza dell'astrazione

Esempi di astrazione...

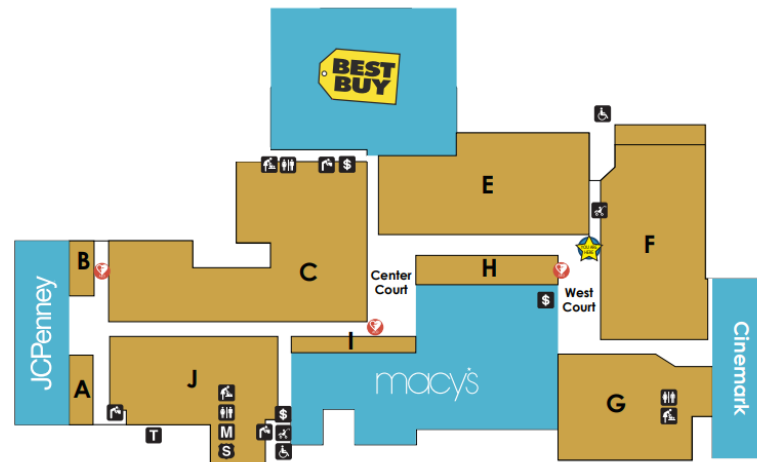
- Una piantina stradale rappresenta una astrazione di una città.



L'importanza dell'astrazione

Esempi di astrazione...

- Una piantina stradale rappresenta una astrazione di una città.

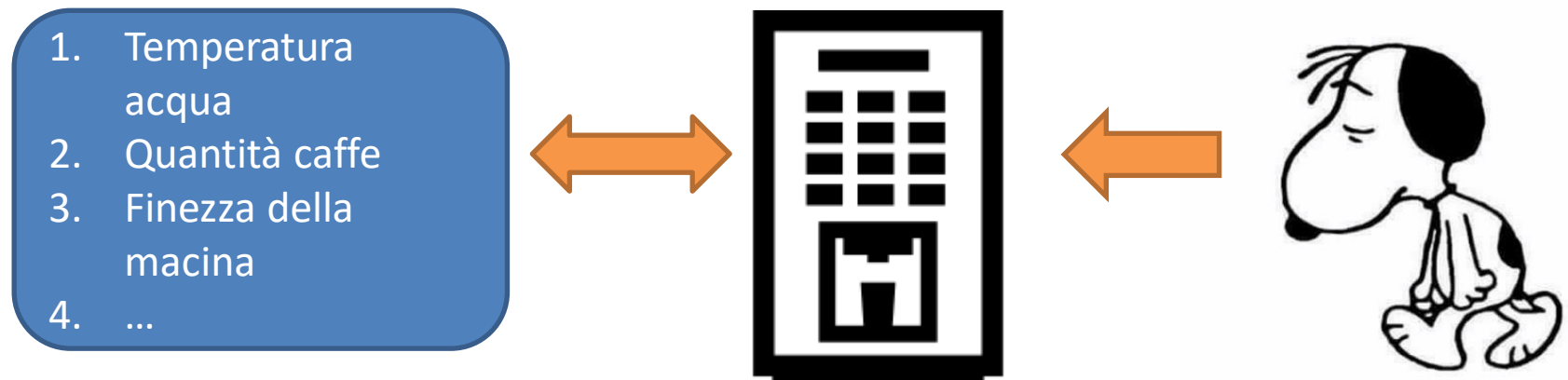


L'importanza dell'astrazione

Esempi di astrazione...

- Una piantina stradale rappresenta una astrazione di una città.
- Come faccio per avere un caffè ?

I need Coffee ☕



L'importanza dell'astrazione

L'**astrazione** è un procedimento che consente di semplificare la realtà che vogliamo modellare. La semplificazione avviene concentrando l'attenzione solo sugli elementi importanti del sistema complesso che stiamo considerando.

L'importanza dell'astrazione

Si tratta di un concetto fondamentale nella programmazione ad oggetti.

Gli oggetti in un linguaggio OOP forniscono la funzionalità di astrarre, cioè di nascondere i dettagli implementativi interni.

Quando si creano dei programmi mediante un linguaggio ad oggetti, la capacità di astrarre, cioè la capacità di semplificare delle entità complesse in ***oggetti caratterizzati dalle caratteristiche e dalle funzionalità essenziali*** per gli scopi preposti, può risultare determinante.

Programmazione procedurale vs. OOP

Nello sviluppo software, usando la metodologia della **programmazione procedurale**, l'interesse principale è rivolto alla sequenza di operazioni da svolgere: si crea un modello indicando le procedure da eseguire in maniera sequenziale per arrivare alla soluzione.

Lo spostamento di attenzione dalle procedure agli oggetti ha portato all'introduzione della **programmazione ad oggetti**. Gli oggetti sono intesi come entità che hanno un loro stato e che possono eseguire certe operazioni.

L'algoritmo perde importanza a vantaggio del concetto di **sistema**.

Programmazione procedurale vs. OOP

Un **algoritmo** è un insieme di istruzioni che a partire dai dati di input permettono di ottenere i risultati di output.

Un algoritmo deve essere riproducibile, deve avere una durata finita e non deve essere ambiguo. Il modo di programmare pone attenzione sulla **sequenza di esecuzione**.

Un **sistema** è una parte del mondo che si sceglie di considerare come un intero, composto da **componenti**. Ogni componente è caratterizzata da **proprietà** rilevanti, e da **azioni** che creano interazioni tra le proprietà e le altre componenti.

Programmazione procedurale vs. OOP

I linguaggi procedurali hanno dei limiti nel creare componenti software riutilizzabili.

I programmi sono fatti da funzioni, che rappresentano codice riutilizzabile, ma che spesso fanno riferimento a headers e/o variabili globali che devono essere importate insieme al codice delle funzioni.

I linguaggi procedurali non si prestano bene alla modellazione di concetti ad alti livelli di astrazione, utili per rappresentare entità complesse che interagiscono in un sistema reale.

In altre parole, i linguaggi procedurali separano le strutture dati e gli algoritmi

Headers
Variabili globali
$f()$
$g()$
$h()$
...
$x()$

Programmazione procedurale vs. OOP

Programmazione procedurale

Problema complesso



Scomposizione in
procedure

Programmazione ad oggetti

Sistema complesso



Scomposizione in
entità interagenti
(oggetti)

Programmare ad oggetti (OOP)

Esempio: interfaccia grafica (GUI) di un PC

Componenti

Finestre (proprietà: dimensione, posizione)

Bottoni (proprietà: colore, testo)

Correlazioni

Premendo un bottone si può aprire una finestra (e quindi definire la sua posizione e la sua dimensione)

Programmare ad oggetti (OOP)

Possiamo individuare tre fasi “Object-Oriented” (OO):

- Analisi (OOA): identificazione dei requisiti funzionali, delle classi e delle loro relazioni logiche.
- Design (OOD): specifica delle gerarchie tra classi, e delle loro interfacce e comportamenti.
- Programmazione (OOP): implementazione del design, test ed integrazione.

La OOP è il momento in cui si scrive effettivamente il codice.

Programmare ad oggetti (OOP)

La metodologia OO è un modo di pensare al problema in termini di sistema, quindi parte dall'analisi del problema e dalla progettazione della sua soluzione.

Durante la fase di analisi si crea un **modello del sistema**, individuando gli **elementi** di cui è formato e i **comportamenti** che devono avere.

In questa fase non interessano le modalità con le quali i comportamenti vengono implementati, ma soltanto gli **oggetti** che compongono il sistema e **le interazioni tra essi**.

Programmare ad oggetti (OOP)

È importante creare programmi che siano **flessibili**, ovvero facili da estendere.

Durante il **design**, lo scopo è avere sw che sia **altamente coeso** ma allo stesso tempo con un accoppiamento lasco (**loose coupling**), ovvero che un cambiamento su una componente non implichi modifiche su un'altra.

Questo si ottiene **decomponendo il problema** in moduli, determinando le relazioni tra essi, identificando dipendenze e forme di comunicazione.

Programmare ad oggetti (OOP)

L'elemento base della OOP è l'**oggetto**.

Un oggetto può essere definito elencando sia le sue caratteristiche, sia il modo con cui interagisce con l'ambiente esterno, cioè i suoi comportamenti.

- Le **caratteristiche** rappresentano gli elementi che caratterizzano l'oggetto, utili per descrivere le sue proprietà e definirne lo stato.
- I **comportamenti** rappresentano le funzionalità che l'oggetto mette a disposizione: chi intende utilizzare l'oggetto deve attivare i comportamenti dell'oggetto stesso

Programmare ad oggetti (OOP)

Una **classe** incapsula sia le **caratteristiche** (attributi) sia i **comportamenti** (metodi) degli oggetti che rappresenta.

Inoltre fornisce una **interfaccia pubblica** per poter utilizzare (interagire con) gli oggetti definiti dalla classe.

In altre parole, la OOP combina **strutture dati e algoritmi** in entità software “impacchettate” dalla definizione di una classe.

Programmare ad oggetti (OOP)

Esempio: analizziamo l'oggetto "automobile"

Caratteristiche: ...

Programmare ad oggetti (OOP)

Esempio: analizziamo l'oggetto "automobile"

Caratteristiche: velocità, colore, numero di porte, livello del carburante, posizione della marcia

Programmare ad oggetti (OOP)

Esempio: analizziamo l'oggetto "automobile"

Caratteristiche: velocità, colore, numero di porte, livello del carburante, posizione della marcia

Comportamenti:

Programmare ad oggetti (OOP)

Esempio: analizziamo l'oggetto "automobile"

Caratteristiche: velocità, colore, numero di porte, livello del carburante, posizione della marcia

Comportamenti: accelera, fermati, gira (a destra o sinistra), cambia marcia, rifornisciti

Programmare ad oggetti (OOP)

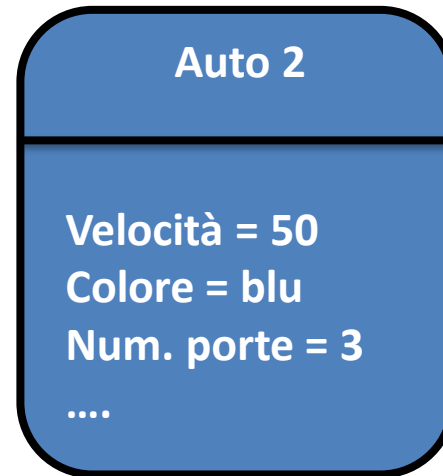
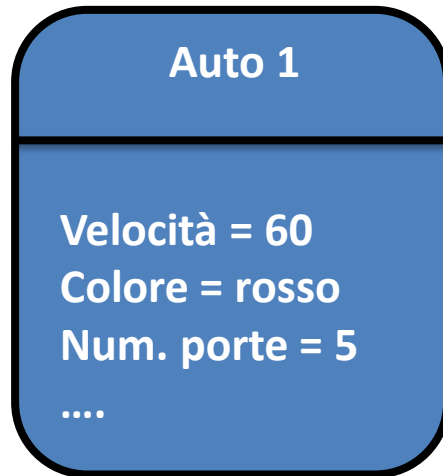
Esempio: analizziamo l'oggetto "automobile"

Caratteristiche: velocità, colore, numero di porte, livello del carburante, posizione della marcia

Comportamenti: accelera, fermati, gira (a destra o sinistra), cambia marcia, rifornisciti

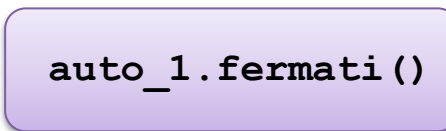
Chi intende utilizzare questo oggetto agisce **attivando i suoi comportamenti**, questi possono concretizzarsi con delle azioni o con il **cambiamento dello stato** dell'oggetto cioè delle sue caratteristiche.

Programmare ad oggetti (OOP)



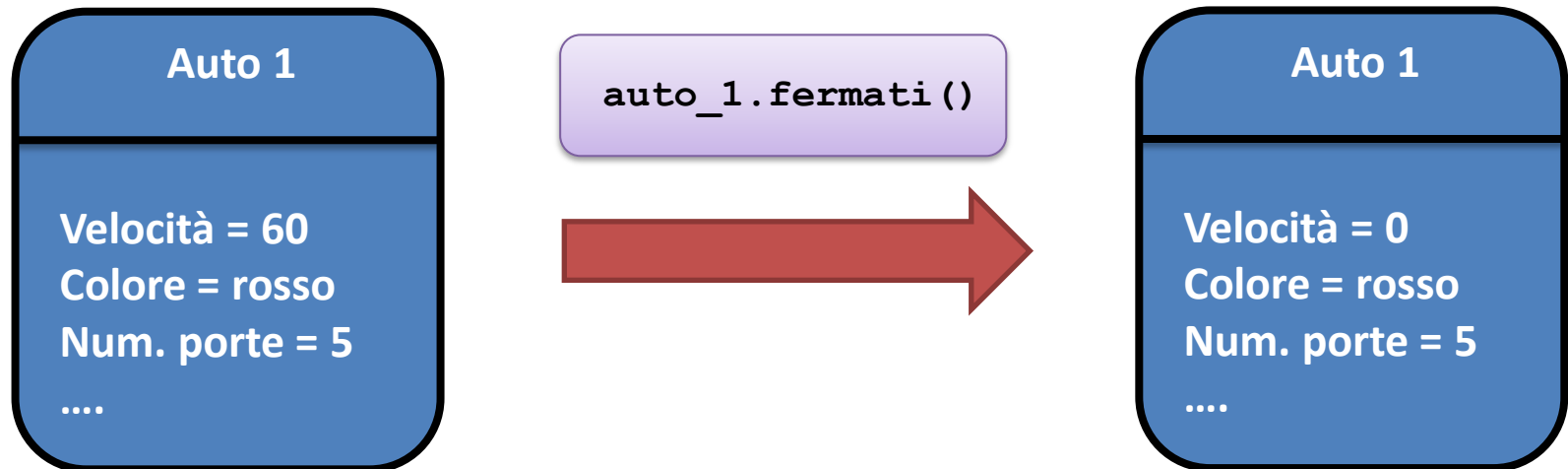
Programmare ad oggetti (OOP)

Mediante il metodo “fermati()” posso modificare lo stato dell’oggetto `auto_1`.



Programmare ad oggetti (OOP)

Mediante il metodo “fermati()” posso modificare lo stato dell’oggetto `auto_1`.



Programmare ad oggetti (OOP)



Programmare ad oggetti (OOP)

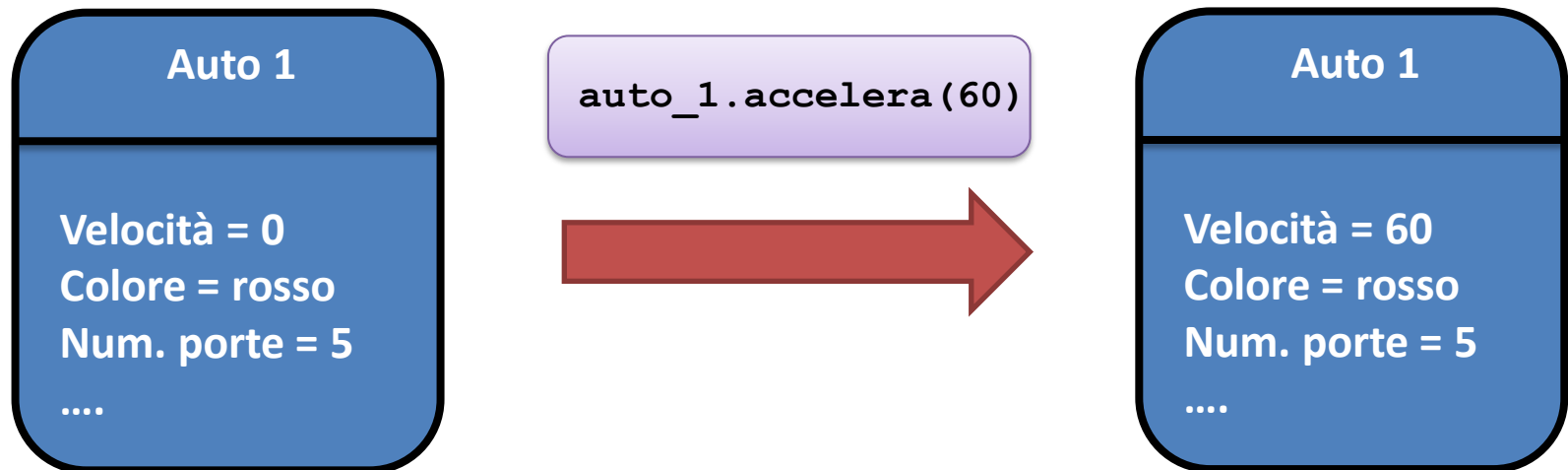
Mediante il metodo “accelera()” posso modificare lo stato dell’oggetto auto_1.



```
auto_1.accelera(60)
```

Programmare ad oggetti (OOP)

Mediante il metodo “accelera()” posso modificare lo stato dell’oggetto `auto_1`.



Programmare ad oggetti (OOP)

Un **oggetto** quindi è formato da **attributi** e **metodi**.

Un **programma ad oggetti** è caratterizzato dalla presenza di **tanti oggetti che comunicano** e interagiscono tra loro (**modello di sistema**).

Nella OOP l'interazione tra oggetti avviene con un meccanismo chiamato **scambio di messaggi**. Un oggetto, inviando un messaggio ad un altro oggetto, può richiederne l'esecuzione di un metodo.

Programmare ad oggetti (OOP)

Adesso che sappiamo cosa significa la programmazione OO e cos'è un oggetto, vediamo ora le caratteristiche fondamentali della OOP che la rendono così importante:

- Incapsulamento
- Interfaccia di un oggetto
- Classe
- Ereditarietà
- Polimorfismo
- Collegamento dinamico

Programmare ad oggetti (OOP)

Il termine ***incapsulamento*** indica la proprietà degli oggetti di incorporare al loro interno sia gli attributi che i metodi, cioè le caratteristiche ed i comportamenti dell'oggetto.

Si dice che gli attributi e i metodi sono *incapsulati* nell'oggetto. In questo modo tutte le informazioni utili che riguardano un oggetto sono ben localizzate.

L'incapsulamento non va confuso con l'***information hiding***, che consiste nel nascondere all'esterno i dettagli implementativi dei metodi di un oggetto.

Programmare ad oggetti (OOP)

Riassumendo possiamo dire che:

Un **oggetto** è costituito da un insieme di **metodi** e **attributi incapsulati** nell'oggetto. Gli oggetti interagiscono sfruttando i **messaggi**, che costituiscono **l'interfaccia** dell'oggetto. L'interfaccia non consente di vedere come sono implementati i metodi, ma permette il loro utilizzo.

Le classi ci permettono di definire dei **tipi di dati astratti** (TDA).

Programmare ad oggetti (OOP)

Non sempre occorre partire dal nulla nel costruire una classe, soprattutto se si dispone già di una classe che è simile a quella che si vuole costruire. In questo caso si può pensare di **estendere** la classe già esistente per adattarla alle nostre necessità.

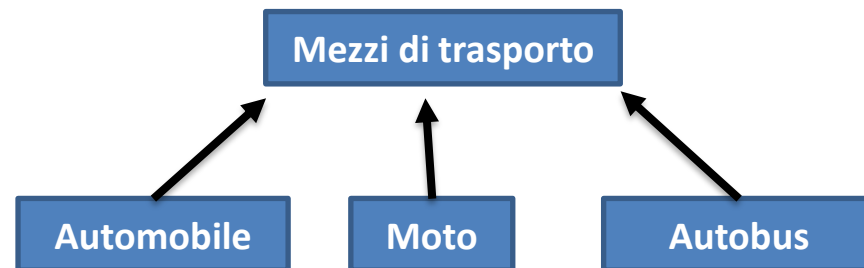
L'**ereditarietà** è lo strumento che permette di costruire nuove classi utilizzando quelle già sviluppate.

Quando una classe viene creata in questo modo, riceve tutti gli attributi ed i metodi della classe generatrice (li eredita). La classe generata sarà quindi costituita da tutti gli attributi e i metodi della classe generatrice più tutti quelli nuovi che saranno definiti.

Programmare ad oggetti (OOP)

La classe che è stata derivata prende il nome di **sottoclasse**, mentre la classe generatrice si chiama **sopraclasse**.

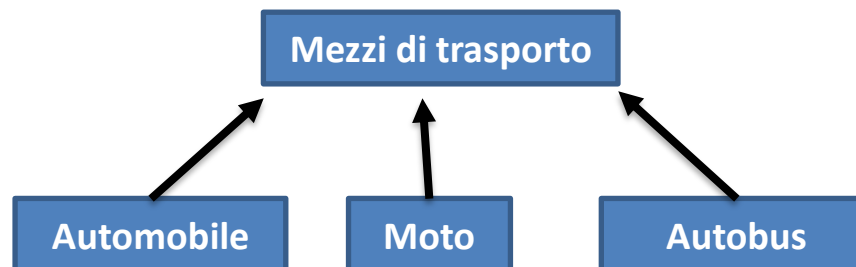
Queste relazioni individuano una gerarchia che si può descrivere usando un **grafo di gerarchia**.



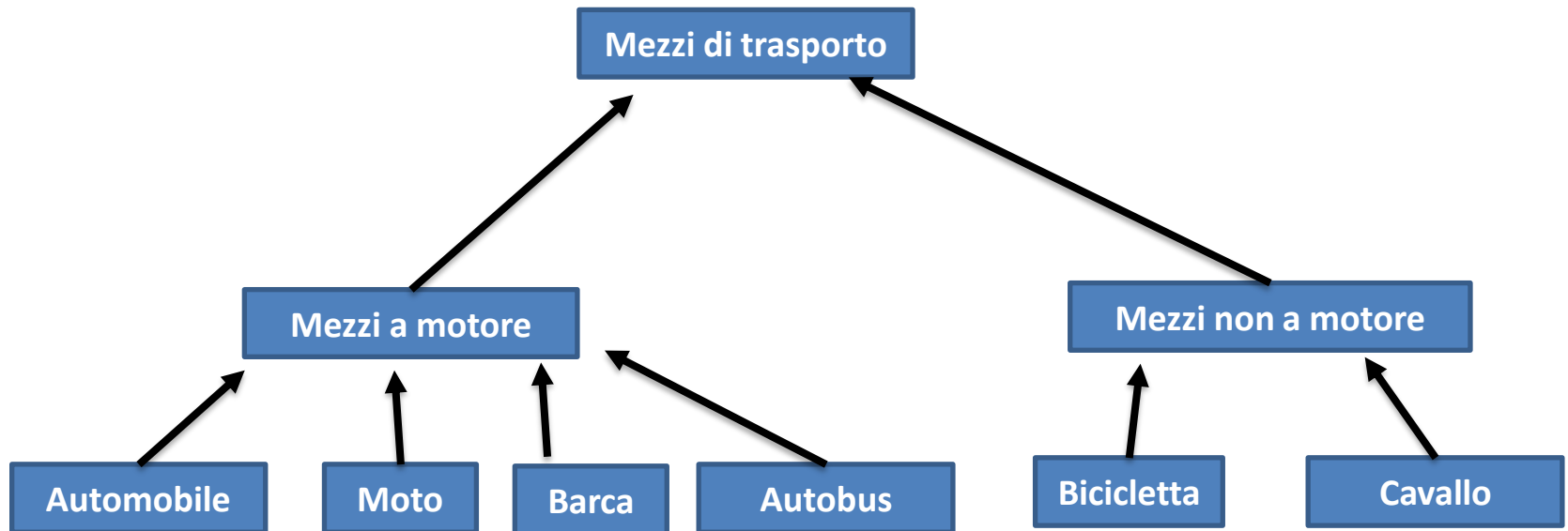
Programmare ad oggetti (OOP)

La nuova classe si differenzia dalla sopraclasse in due modi:

- Per estensione: aggiungendo nuovi attributi e metodi
- Per ridefinizione: modificando i metodi ereditati, specificando una implementazione diversa di un metodo (override, overload)

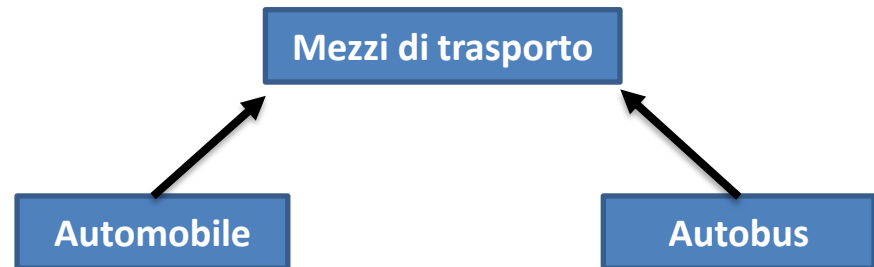


Programmare ad oggetti (OOP)

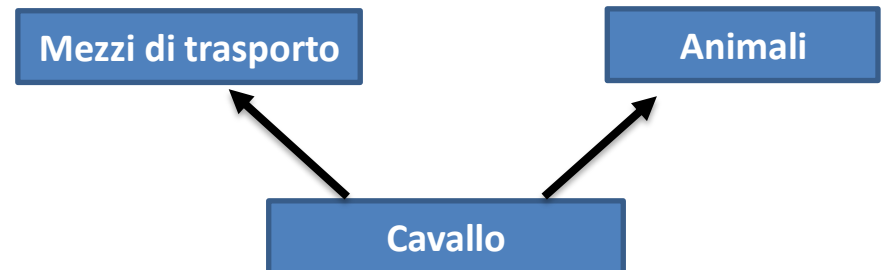


Programmare ad oggetti (OOP)

Ereditarietà singola



Ereditarietà multipla



Programmare ad oggetti (OOP)

Considerando una relazione di ereditarietà, le sottoclassi hanno la possibilità di ridefinire i metodi ereditati (mantenendo lo stesso nome) oppure lasciarli inalterati perché già soddisfacenti.

Il **polimorfismo** indica la possibilità per i metodi di assumere forme, cioè implementazioni, diverse all'interno della gerarchia delle classi.

Esempio: tutti i veicoli a motore possiedono il metodo "accelera". Le sottoclassi "automobile" e "moto" è probabile che lo ridefiniscano per adeguarlo alle particolari esigenze (es. pedale vs. manopola).

Programmare ad oggetti (OOP)

Durante l'esecuzione del programma, un'istanza della classe "veicoli a motore" può rappresentare sia una "automobile" che una "moto".

Quando viene richiesta l'attivazione del metodo "accelera" è importante garantire che, tra tutte le implementazioni, venga scelta quella corretta.

Il **collegamento dinamico** è lo strumento utilizzato per la realizzazione del polimorfismo. È dinamico perché l'associazione tra l'oggetto e il metodo corretto da eseguire è effettuata a ***run-time***, cioè durante l'esecuzione del programma.

Programmare ad oggetti (OOP)

Abbiamo visto le proprietà e le caratteristiche principali della OOP. Questo paradigma di programmazione ha offerto un modo nuovo e potente per scrivere programmi.

Per programmare ad oggetti serve una nuova metodologia con cui affrontare i problemi, che possiamo riassumere nel seguente modo:

- Identificare gli oggetti che caratterizzano il modello del problema
- Definire le classi indicando gli attributi e i metodi
- Stabilire come gli oggetti interagiscono con gli altri



UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI
MATEMATICA E INFORMATICA

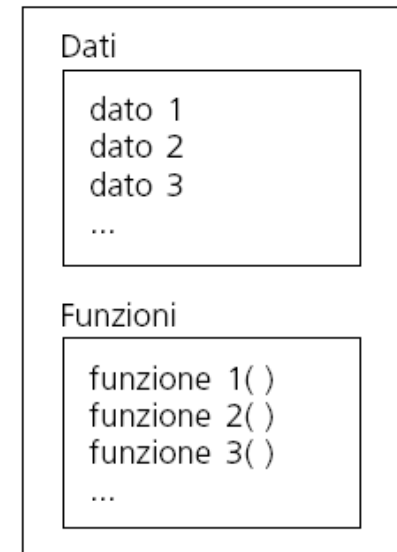
OOP in C++

Alessandro Ortis

ortis@dmf.unict.it
www.dmf.unict.it/ortis/

Le classi

- Classe: insieme di oggetti che condividono una struttura ed un comportamento
- Contiene la specifica dei dati che descrivono l'oggetto che ne fa parte, insieme alla descrizione delle azioni che l'oggetto stesso è capace di eseguire
- in C++ questi dati si denominano *attributi* o *variabili*, mentre le azioni si dicono *funzioni membro* o *metodi*
- Le classi definiscono ***tipi di dato personalizzati*** in funzione dei problemi da risolvere,
 - ciò facilita scrittura e comprensione delle applicazioni;
- Possono ***separare l'interfaccia dall'implementazione***;
 - solo il programmatore della classe conoscerà i dettagli implementativi,
 - l'utilizzatore deve soltanto conoscere l'interfaccia



Definizione di una classe

- Due parti:
 - *dichiarazione*: descrive i dati e l'interfaccia (cioè le "funzioni membro", anche dette "metodi")
 - *definizioni dei metodi*: descrive l'implementazione delle funzioni membro

```
class NomeClasse          // Identificatore valido
{
    dichiarazioni dei dati      // attributi
    definizione delle funzioni // metodi
};
```

- Attributi: variabili semplici (interi, strutture, arrays, float, ecc.)
- Metodi: funzioni semplici che operano sugli attributi (*dati*)

Specificatori di accesso

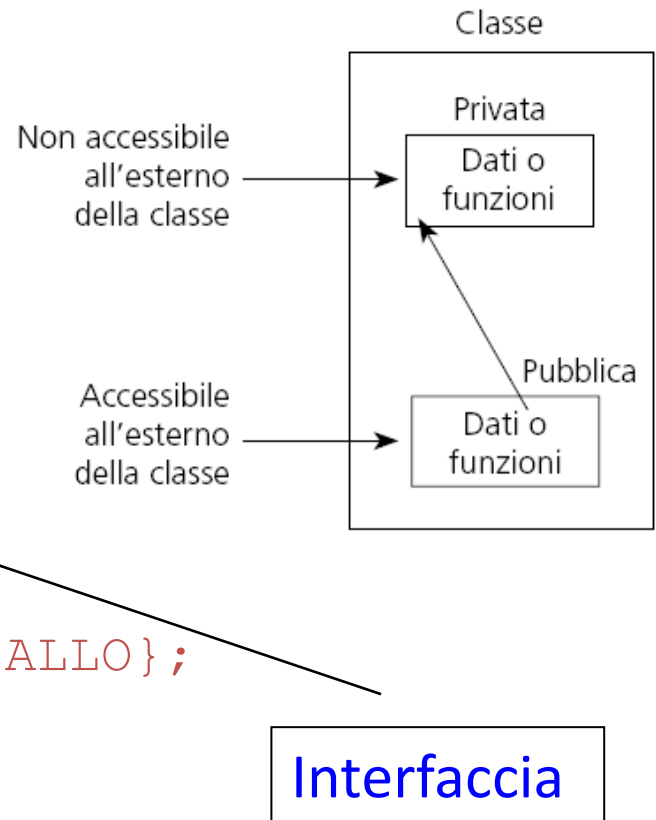
- Per default, i membri di una classe sono nascosti all'esterno, cioè, i suoi dati ed i suoi metodi sono *privati*
- E' possibile controllare la *visibilità* esterna mediante ***specificatori d'accesso***:
 - la sezione `public` contiene membri a cui si può accedere dall'esterno della classe
 - la sezione `private` contiene membri ai quali si può accedere solo dall'interno
 - ai membri che seguono lo specificatore `protected` si può accedere anche da metodi di classi *derivate* della stessa

```
class NomeClasse
{
    public:
        Sezione pubblica    // dichiarazione di membri pubblici
    protected:
        Sezione protetta    // dichiarazione di membri protetti
    private:
        Sezione privata      // dichiarazione di membri privati
};
```

Information hiding

- Questa caratteristica della classe si chiama *occultamento di dati (information hiding)* ed è una proprietà dell'OOP
- limita molto gli errori rispetto alla programmazione strutturata

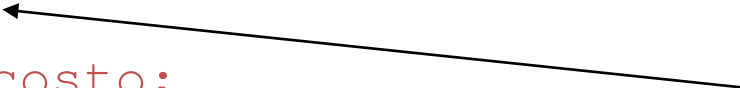
```
class Semaforo
{
    public:
        void cambiareColore();
        //...
    private:
        enum Colore {VERDE, ROSSO, GIALLO};
        Colore c;
};
```



Regole pratiche

- le dichiarazioni dei metodi (i.e., intestazioni delle funzioni), normalmente, si collocano nella sezione pubblica
- le dichiarazioni dei dati (attributi), normalmente, si mettono nella sezione privata
- E' indifferente collocare prima la sezione pubblica o quella privata;
 - Meglio collocare la sezione pubblica prima per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica
- `public` e `private` seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private;
 - una classe può avere varie sezioni pubbliche e private
- L'interfaccia deve essere pubblica
 - Altrimenti non può essere invocata dal programma!

```
class Prova
{
    private:
        float costo;
        char nome[20];
    public:
        void calcolare(int);
};
```



`private` non è necessario
ma è utile per evidenziare
l'occultamento

Per usare una classe bisogna sapere

.Nome

- Tipicamente definito in un header file (con lo stesso nome della classe)

.Dove è definita

.Che operazioni supporta

- Nelle definizioni delle classi si collocano (tipicamente) solo le intestazioni dei metodi
- Le definizioni dei metodi stanno in un file di *implementazione* (estensione .cpp)

Oggetti

- definita una classe, possono essere generate *istanze* della classe, cioè *oggetti*

```
nome_classe    identificatore ;
```

```
Rettangolo r;  
Semaforo S;
```

- un oggetto sta alla sua classe come una variabile al suo tipo
- quello che nelle struct era l'operatore di accesso al campo (.), qui diventa l'operatore *di accesso* al membro

```
Punto p;  
p.FissareX(100);  
cout << " coordinata x è " << p.LeggereX();
```

Oggetti

- Gli oggetti (come le strutture) possono essere copiati
- C++ fa una copia bit a bit di tutti i membri
- Tutti i membri presenti nell'area dati dell'oggetto originale vengono copiati nell'oggetto destinatario

```
Rettangolo r1;
```

```
...
```

```
Rettangolo r2;
```

```
r2=r1;
```

Dati membro

- possono essere di qualunque tipo valido, tranne il tipo della classe che si sta definendo.
- Un dato membro (attributo) ha un nome (o identificatore) e un tipo; mantiene un valore di uno specifico tipo (tipo base o altra classe).
- Convenzione sui nomi: si usa la notazione a cammello. I nomi delle variabili sono parole singole (sostantivi), oppure parole composte unendo più parole tra loro, ma lasciando le iniziali maiuscole.

```
double coeffAngolare;  
int anniPersona;  
...
```

Funzioni membro

- possono essere sia dichiarate che definite all'interno delle classi; la definizione consiste di quattro parti:
 - il tipo restituito dalla funzione
 - il nome della funzione
 - la lista dei parametri formali (eventualmente vuota) separati da virgole
 - il corpo della funzione racchiuso tra parentesi graffe
- le tre prime parti formano il **prototipo** (o signature) della funzione che *deve essere definito* dentro la classe,
 - il corpo della funzione può essere definito altrove

```
class Quadrato {
public:
    double calcola_area(); // dichiarazione prototipo (definito altrove)
    double calcola_perimetro () // definizione
        {return lato*4; } // funzione
private: // membri privati
    double lato;
};

double Quadrato::calcola_area(){ return lato*lato;}
```

Funzioni membro

- La definizione di funzioni dichiarate in una classe deve contenere il riferimento alla classe

```
tipo_restituito Nome_Classe :: Nome funzione  
(lista parametri)  
{  
    corpo della funzione  
}
```



```
double Quadrato::calcola_area() {  
    return lato*lato;  
}
```

Chiamate a funzioni membro

- i metodi di una classe s'invocano così come si accede ai dati di un oggetto, tramite l'operatore punto (.) con la seguente sintassi:

nomeOggetto.nomeFunzione (valori dei parametri)

```
class Demo
{
private:
    // ...
public:
    void funz1 (int P1)
        {...}
    void funz2 (int P2)
        {...}
};
Demo d1, d2;           // definizione degli oggetti d1 e d2
...
d1.funz1(2005);
d2.funz1(2010);
```

**Alcuni linguaggi chiamano messaggi
le invocazioni a funzioni membro**

Tipi di funzioni membro

- Costruttori e distruttori
 - Invocati automaticamente alla creazione e alla distruzione di oggetti
- Selettori
 - Restituiscono valori di membri dato
- Modificatori
 - Modificano i valori di membri dato
- Operatori
 - definiscono operatori standard in C++
- Iteratori
 - elaborano collezioni di oggetti (es. array)

Funzioni *inline* e *offline*

- i metodi definiti nella classe sono funzioni in linea; per funzioni grandi è preferibile codificare nella classe solo il prototipo della funzione
- nella definizione *fuori linea* della funzione bisogna premettere il nome della classe e l'*operatore di risoluzione di visibilità* ::;

```
class Punto {  
public:  
    void fissareX(int valx);  
private:  
    int x;  
    int y;  
};  
  
void Punto::fissareX(int valx)  
{  
    x = valx;  
}
```

Nella dichiarazione il nome dei parametri può essere omesso.

Header files ed intestazioni di classi

- il codice sorgente di una classe si colloca normalmente in un file indipendente con lo stesso nome della classe ed estensione .cpp
- le dichiarazioni si collocano normalmente in header files indipendenti da quelli che contengono le implementazioni dei metodi

Costruttori

- Può essere conveniente che un oggetto si possa auto-inizializzare all'atto della sua creazione, senza dover effettuare una successiva chiamata ad una sua qualche funzione membro
- un *costruttore* è un metodo di una classe che viene automaticamente eseguito all'atto della creazione di un oggetto di quella classe
- ha lo stesso nome della propria classe e può avere qualunque numero di parametri ma non restituisce alcun valore

```
class Rettangolo
{
    private:
        int base;
        int altezza;

    public:
        Rettangolo(int base, int altezza); // Costruttore
        // definizioni di altre funzioni membro
};
```

Definizione oggetto con costruttore

- quando si definisce un oggetto, si passano i valori dei parametri al costruttore utilizzando la sintassi di una normale chiamata di funzione:

```
Rettangolo rect(25, 75); // rect è ISTANZA di Rettangolo
```

```
Rettangolo* nr = new Rettangolo(25, 75); // nr punta  
// una nuova ISTANZA di Rettangolo
```

- un costruttore senza parametri si chiama *costruttore di default*;
 - inizializza i membri dato assegnandogli valori di default
- C++ crea automaticamente un costruttore di default quando non vi sono altri costruttori, esso non inizializza i membri dato della classe a valori predefiniti
- un *costruttore di copia* è creato automaticamente dal compilatore quando:
 - si passa un oggetto per valore ad una funzione (si costruisce una copia locale dell'oggetto)
 - si definisce un oggetto inizializzandolo ad un oggetto dello stesso tipo

Definizione oggetto con costruttore


Esistono diversi modi per inizializzare gli attributi di un oggetto.

```
class MyClass{
    public:
        // Costruttore di default
        MyClass(){}
        // lista di inizializzazione
        MyClass(int a, char b): x(a), c(b) {}
        MyClass(int a): y(a*2) {}
        // costruttore con delega
        MyClass(char b): MyClass(10, b){}
        void printAll();
    private:
        int x = 18; // inizializzazione
        char c {'M'}; // iniz. Uniforme
        const int y = 0;
};
```

Definizione oggetto con costruttore

Esistono diversi modi per inizializzare gli attributi di un oggetto.

```
class MyClass{
    public:
        // Costruttore di default
        MyClass(){}
        // lista di inizializzazione
        MyClass(int a, char b): x(a), c(b) {}
        MyClass(int a): y(a*2) {}
        // costruttore con delega
        MyClass(char b): MyClass(10, b){}
        void printAll();
    private:
        int x = 18; // inizializzazione
        char c {'M'}; // iniz. Uniforme
        const int y = 0;
};
```



Funzioni di
conversioni
implicite.

Qualificatore `const`

Il qualificatore `const` indica che un tipo è costante ed è utilizzabile in diversi ambiti, allo scopo di evitare la modifica di informazioni marcate come costanti.

```
const double pi = 3.14;
```

Puntatori e qualificatore `const`

```
int a = 22;  
  
int* p1 = &a; // niente e' costante  
const int* p2 = &a; // dato costante  
int* const p3 = &a; // puntatore costante  
const int* const p4 = &a; // tutto costante
```

Parametri di funzioni `const`

```
void funzione(const int& a) {  
    // qualcosa che non modifica a  
}
```

Qualificatore `const`

Le funzioni membro `const` non possono modificare gli attributi dell'oggetto di appartenenza.

```
class MyClass{
    int x;
    ...
    int myFunction() const;
};

int MyClass::myFunction() const {
    // non posso modificare x
}
```

Valori di ritorno `const`

```
class MyClass{
    int x= 100;
    public:
        int const & goodGetX(){ return x;}
};
```

Distruttore

- si può definire anche una funzione membro speciale nota come *distruttore*, che viene chiamata automaticamente quando si distrugge un oggetto
- il distruttore ha lo stesso nome della sua classe preceduto dal carattere ~
- neanche il distruttore ha tipo di ritorno ma, al contrario del costruttore, non accetta parametri e *non* ve ne può essere più d'uno

```
class Demo
{
private:
    int dati;
public:
    Demo()    {dati = 0;}           // costruttore
    ~Demo()   {}                   // distruttore
};
```

- serve normalmente per liberare la memoria assegnata dal costruttore
- se non si dichiara esplicitamente un distruttore, C++ ne crea automaticamente uno vuoto

Membri statici

Un membro statico è associato alla classe anziché con un oggetto (istanza di una classe). Questo significa che ne esiste una copia unica per tutte le istanze di quella classe nel caso di un attributo statico.

Nel caso di un metodo statico, significa che è possibile invocare quel metodo senza aver istanziato alcun oggetto.

```
class Point {  
public:  
    static int n;  
  
    Point(): x(0), y(0) {n++;}           // costruttore  
    ~Point() { n--; }                   // distruttore  
  
    static float distance(Point a, Point b) { //... }  
  
private:  
    int x,y;  
};
```

Friend e incapsulamento

A volte può essere utile consentire l'accesso a membri privati anche a funzioni o metodi di altre classi. Questo può essere utile nell'overloading di operatori di input/output.

```
class Point {  
public:  
    // dichiarazione di amicizia  
    friend bool operator==(Point a, Point b);  
private:  
    int x,y;  
};  
  
bool operator==(Point a, Point b){  
    if ( (a.x != b.x) || (a.y) != (b.y) ) return false;  
    else return true;  
}  
  
int main(){ Point p,q;  
if (p == q) cout << 'p e q sono uguali' << endl; }
```

Friend e incapsulamento

```
bool operator==(Point a, Point b){  
    if ( (a.x != b.x) || (a.y) != (b.y) ) return false;  
    else return true;  
}
```

Alternativa senza friend

```
class Point {  
public:  
    // dichiarazione come funzione membro  
    bool operator==(Point b);  
private:  
    int x,y;  
};  
  
bool Point::operator==(Point b){  
    if ( (this->x != b.x) || (this->y) != (b.y) ) return  
false;  
    else return true;  
}
```

Friend e incapsulamento

La funzione `operator==` non è un membro della classe `Point`, eppure può accedere ai suoi membri privati. L'uso del qualificatore `friend` consente di uniformare l'interfaccia di I/O della nostra classe allo standard senza violare il principio di incapsulamento. Infatti, anche se le funzioni `friend` non sono membri di classe, la loro dichiarazione deve apparire nella definizione della classe. Questo ci consente di rimanere in totale controllo dell'interfaccia, rendendo impossibili alterazioni dall'esterno mediante la definizione di funzioni `friend`.

```
class Point {  
    ... // dichiarazione di amicizia  
        friend bool operator==(Point a, Point b);  
    ...  
};
```

```
bool operator==(Point a, Point b){  
    if ( (a.x != b.x) || (a.y) != (b.y) ) return false;  
    else return true;    }
```

Sovraccaricamento di metodi e operatori

- anche le funzioni membro possono essere sovraccaricate, ma soltanto nella loro propria classe
- Seguono le stesse regole utilizzate per sovraccaricare funzioni ordinarie:
 - due funzioni membro sovraccaricate non possono avere lo stesso numero e tipo di parametri
 - l'*overloading* permette di utilizzare uno stesso nome per più metodi che si distingueranno solo per i parametri passati all'atto della chiamata

```
class Prodotto
{
public:
    int prodotto (int m, int n);           // metodo 1
    int prodotto (int m, int p, int q);    // metodo 2
    int prodotto (float m, float n);       // metodo 3
    int prodotto (float m, float n, float p); // metodo 4
}
```

Sovraccaricamento di metodi e operatori

```
class Array{
public:
    Array(int size=10); // costruttore
    // l'operatore == non puo' modificare nulla
    bool operator ==(const Array& right) const;
    int& operator[] (int index);
    ...
    int size;          int* data;
}

bool Array::operator==(const Array& right) const{
    if(size!= right.size) return false;
    for(int i=0;i<size; i++)
        if(data[i] != right.data[i])
            return false;
    return true;
}

int& Array::operator[] (int index){
    return data[index];
}
```

Gestione delle eccezioni

In molti casi possiamo prevenire gli errori a runtime dei nostri programmi quando si verificano situazioni anomale relativamente alla logica del problema.

```
class Tempo {  
private:  
    int ore;        // 0 - 23  
    int minuti;     // 0 - 59  
    int secondi;    // 0 - 59  
    ...  
void setOre(int h) {  
    if (h >= 0 && h <= 23)  
        ore = h;  
    else {  
        cout << "Errore: valori di ore validi 0-23." << endl;  
        exit(1); // Termina il programma  
    }  
};
```

Gestione delle eccezioni

Il metodo `setOre()` assegna `h` all'ora se `h` è un'ora valida. Altrimenti, usiamo la funzione di gestione delle eccezioni C++ per lanciare una eccezione del tipo `invalid_argument`.

```
#include <stdexcept> // Necessario per gestire le eccezioni

class Tempo {
private:
    int ore;        // 0 - 23
    int minuti;     // 0 - 59
    int secondi;    // 0 - 59
    ...
void setOre(int h) {
    if (h >= 0 && h <= 23)
        hour = h;
    else
        throw invalid_argument("Errore: valori di ore validi
0-23.");
}
```


Gestione delle eccezioni

Ciò consente al chiamante di catturare l'eccezione e di elaborare correttamente la condizione anomala.

```
int main() {  
  
    Time t;  
  
    try {  
        t.setOre(100);  
    }  
    catch (invalid_argument& ex) {  
        cout << "Eccezione: " << ex.what() << endl;  
    }  
  
    return 0;  
} // fine del main
```

Esercizio

Definire una classe Rettangolo con i seguenti requisiti funzionali:

- Attributi base e altezza privati, ma accessibili tramite getter/setter.
- Il costruttore prende la base e l'altezza come parametri, in alternativa imposta due valori di default.
- Implementare delle funzioni membro pubbliche per il calcolo di area, perimetro e diagonale
- Implementare una funzione membro che verifica se si tratta di un quadrato

Definire un metodo main() dove vengono istanziati alcuni oggetti Rettangolo per testare le funzionalità della classe.

Hint: utilizzare in maniera opportuna i diversi modi per definire un costruttore ed il qualificatore **const**.

Esercizio

Definire una classe `Punto2D` e utilizzarla per definire una classe `Rettangolo` simile a quella precedente. In particolare la nuova classe `Rettangolo` dovrà:

- avere due attributi `top_left` e `bottom_right` di tipo `Punto2D`
- prevedere i seguenti metodi:
 - `contiene(Punto2D p)` restituisce vero se `p` si trova dentro l'area del rettangolo chiamante
 - `contiene(Rettangolo r)` restituisce vero se `r` è contenuto nel rettangolo chiamante

Inoltre, definire le classi `Punto2D` e `Rettangolo` in modo tale che sia possibile istanziare due oggetti, uno di tipo `Punto2D` e l'altro di tipo `rettangolo` nel seguente modo:

```
Punto2D p = {10,20}  
Rettangolo B({10,20},{50,10});
```

dove:

- `{10,20}` rappresenta un oggetto `Punto2D` di coordinate `x=10 y=20`
- `{50,10}` rappresenta un oggetto `Punto2D` di coordinate `x=50 y=10`