

IV Oggetti e classi

IV.1 La programmazione ad oggetti

L'acronimo **OOP** in informatica sta per *Object Oriented Programming* (Programmazione Orientata agli Oggetti) e vuole denotare l'insieme delle caratteristiche a fondamento del **paradigma della programmazione ad oggetti**. Storicamente, le prime idee fondanti di questo paradigma sono sorte con il *Simula '67*, un linguaggio di simulazione discreta di scuola scandinava, e sono state poi riscoperte negli USA negli anni '70-'80 con il linguaggio *Smalltalk*. Buona parte del gergo della OOP si deve a questi due linguaggi.

Dalla metà degli anni '80 esiste un generale accordo sulle caratteristiche che devono essere presenti in un linguaggio per poterlo classificare come OOPL, cioè per affermare che mette a disposizione il paradigma della OOP:

1. **insieme di operazioni astratte, associate ad un tipo**
2. **stato locale**
3. **ereditarietà**

La prima caratteristica stabilisce che l'unico modo per agire sugli oggetti di un certo tipo è quello di invocare delle *operazioni* (nel gergo OOP chiamate **metodi**) prestabilite e note agli utenti in modo astratto, cioè indipendente dalla *rappresentazione* prescelta per la loro concreta *implementazione* (realizzazione).

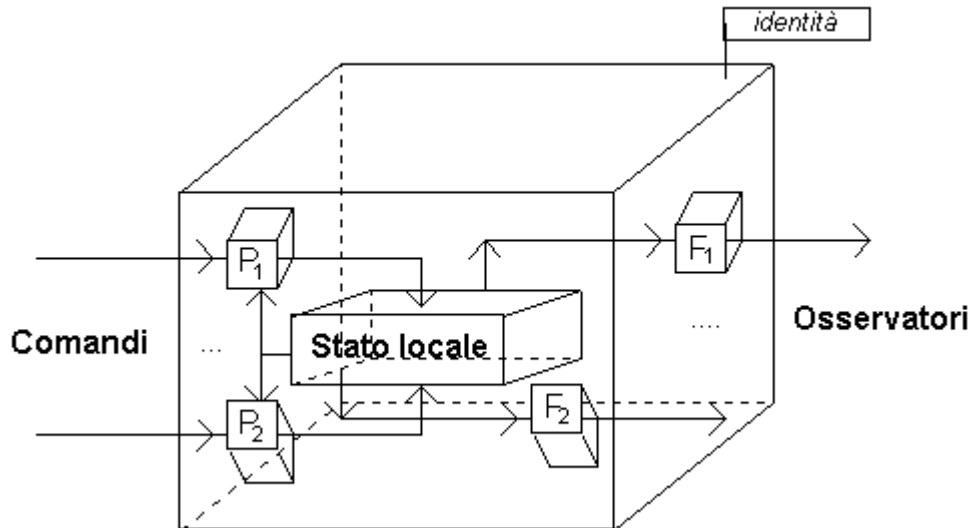
La seconda caratteristica stabilisce che ogni oggetto ha un proprio **stato locale** che può cambiare per effetto dell'esecuzione di alcuni metodi. In pratica il valore dello stato locale ad un certo istante è stabilito dal contenuto di un insieme di celle di memoria associate all'oggetto, dette **campi** o **attributi** dell'oggetto.

L'**ereditarietà** stabilisce infine che il linguaggio deve prevedere un meccanismo per stabilire delle relazioni tra *classi* di oggetti. Questo meccanismo consente in definitiva il **riuso** e il **polimorfismo**, due dei principali fattori alla base del successo della OOP.

Un **oggetto** (software) è un'entità dotata di:

- **identità**, che permette quindi sempre di distinguere un oggetto da un altro (un "numero di serie")
- **stato**, quindi in grado di "ricordare" qualcosa
- **comportamento**, che si traduce nella possibilità di osservare (in tutto o in parte) lo stato e/o di modificare lo stato, tramite l'invocazione dei *metodi* sull'oggetto.

Un modo corretto di pensare ad un oggetto è rappresentato nella seguente figura:



I blocchi P1, P2, ... ed F1, F2, ... rappresentano i "meccanismi interni" dell'oggetto: in pratica sono dei sottoprogrammi (i metodi) che realizzano le operazioni previste dall'oggetto. I metodi vengono spesso divisi in due categorie:

- **comandi**, cioè metodi che modificano lo stato dell'oggetto. Sono *procedure proprie*, cioè che non restituiscono valori ma che producono degli effetti collaterali, generalmente limitati allo stato locale dell'oggetto.
- **osservatori** (*queries*), cioè metodi che si limitano ad osservare lo stato dell'oggetto. Sono *procedure di tipo funzionale* che restituiscono un valore ottenuto come elaborazione (solitamente molto semplice) dello stato corrente dell'oggetto.

Per **creare** gli oggetti la tecnica più seguita è quella di definire prima una **classe**, cioè uno schema di creazione che definisca i possibili stati e i comportamenti, e di invocare un'operazione di **istanziamento della classe**, detta *costruttore* o *creatore*, che determina univocamente l'identità di un oggetto e che ne stabilisce lo *stato iniziale*. Una possibile sintassi di creazione (per linguaggi tipati) potrebbe essere:

variabile **nuovo** [*nome_costruttore* (*parametri_attuali_se_previsti*)]

In pratica una classe sarà definita da un *nome* a cui viene associato un elenco di *costruttori*, *attributi* e *metodi*, che collettivamente chiameremo **proprietà** della classe. Nei linguaggi OOP tipati, la definizione di una classe comporta generalmente anche quella di un tipo con lo stesso nome.

Si ammette che dall'elenco possano mancare i costruttori: in tal caso viene usato implicitamente un costruttore di default (senza parametri).

Per accedere al valore di un attributo o per invocare un metodo su un oggetto la maggior parte dei linguaggi fa uso della **notazione punto**:

espressioneoggetto.nome_attributo
espressioneoggetto.nome_metodo(*eventuali_parametri_attuali_se_previsti*)

Information Hiding. Il concetto di **information hiding** (che potremmo tradurre con "occultamento di informazione") consiste nell'idea che l'utente di un servizio è tenuto a conoscere solo le informazioni strettamente necessarie per l'usufruzione del servizio. Ogni altra informazione può confondere l'utente e/o mettere a rischio l'integrità dell'oggetto stesso.

Tecnicamente si afferma che l'utente deve conoscere solo l' **interfaccia della classe**, cioè il suo nome e l'interfaccia di ciascuna proprietà pubblica.

Alcuni sistemi sono in grado di estrarre/documentare automaticamente l'interfaccia di classe dalla definizione completa di un classe.

In realtà esiste un altro importante vantaggio dell'information hiding: l'implementatore di una classe potrà ritenersi libero di effettuare delle migliorie senza che questo comporti delle modifiche ai programmi clienti, cioè che usino i servizi offerti da quella classe. L'information hiding è quindi determinante non solo per semplificare la vita ai programmatori, ma anche per la manutenzione e il "riuso del software".

In particolare, nella OOP, l'informazione da nascondere è la **rappresentazione** dello stato dell'oggetto, cioè l'insieme degli attributi. A rigore, quindi, *tutti gli attributi dovrebbero essere nascosti*: l'utente deve potervi accedere solo indirettamente attraverso i metodi. Tuttavia, per motivi di efficienza, alcuni linguaggi ad oggetti (tra cui C++, Java e Delphi) permettono l'accesso agli attributi (sia in lettura che in scrittura). Il problema dell'information hiding viene risolto in questi linguaggi permettendo l'accesso solo ad alcuni attributi classificati come **pubblici** (tipicamente saranno quelli presenti in tutte le possibili rappresentazioni). Oltre agli attributi si possono nascondere anche metodi. Tipico è il caso dei metodi ausiliari, cioè utilizzati per implementare i metodi pubblici, e più in generale, di tutti quei metodi la cui esistenza è legata alla rappresentazione scelta.

In generale, *ogni linguaggio propone proprie soluzioni al problema della visibilità*, cioè di come stabilire a quali classi rendere visibili le proprietà (sia attributi che metodi) di una classe. A titolo puramente indicativo possiamo citare la soluzione a tre livelli del C++:

proprietà **pubbliche**: *tutte le classi* possono accedervi
proprietà **protette**: solo le *sottoclassi* possono accedervi
proprietà **private**: *nessuna classe* può accedervi

Infatti, per questioni di flessibilità ed efficienza, si sente la necessità di poter definire diversi "livelli" di visibilità. L'ideale è poter decidere per ogni proprietà, quali sono le classi che possono accedervi. L'ereditarietà, in particolare, pone un problema che occorre risolvere di volta in volta: quali proprietà ereditate di una classe devono essere nascoste alle sue sottoclassi? alcuni linguaggi (ad es. C++ e Java) danno questa risposta: quelle pubbliche e quelle dichiarate appunto come "protette". Altri (ad es. Eiffel) sostengono la tesi che tutto quello che è visibile all'implementatore di una classe deve essere visibile anche agli implementatori delle sue sottoclassi. Inoltre Eiffel non permette la modalità "private", sostenendo il punto di vista secondo cui l'implementatore di una classe non può decidere arbitrariamente che alcuni attributi non siano utili agli implementatori delle sottoclassi.

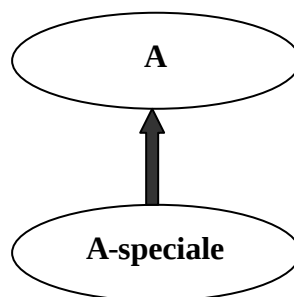
Ereditarietà. L'ereditarietà risulta indispensabile per processi di modellazione molto importanti nella moderna costruzione del software quali la *specializzazione* e la *generalizzazione*. Nella sua forma più semplice, l'**ereditarietà** (*inheritance*) è un meccanismo che consente ad una classe di considerarsi erede di un'altra, detta **classe padre** o genitore (*parent class*), con una dichiarazione che possiamo assumere del tipo:

classe <nome> **eredita da** <classe padre>

...

Così facendo la classe <nome>, detta **classe figlia** (o classe derivata), eredita tutte le proprietà della <classe padre> specificata : tutti gli attributi e i metodi (anche quelli nascosti).

Sono state proposte varie notazioni grafiche per rappresentare una relazione di ereditarietà tra classi. Una delle più semplici è la seguente, in cui la relazione è rappresentata da un arco diretto dalla classe figlia alla classe padre. Rappresentando l'insieme delle relazioni di questo genere si ottiene il **grafo di ereditarietà**



Da un punto di vista pratico si può ereditare da qualsiasi classe, a patto di *non introdurre cicli* nel grafo di ereditarietà. Tuttavia, come suggerito dai nomi delle classi della figura, da un punto di vista metodologico è *opportuno e consigliato ereditare* X da Y solo quando si è certi che tra Y e X esiste una relazione Y **IS-A** X ("Y è un X"), o in altri termini, quando la classe figlia è una **specializzazione** di quella padre.

Oltre a ereditare, la classe figlia può:

- **aggiungere** propri attributi o metodi
- **ridefinire** (**overriding**) metodi ereditati

In questo caso il metodo ereditato (metodo *precursore*) e quello ridefinito hanno lo stesso nome. I linguaggi devono predisporre meccanismi linguistici opportuni per consentire all'implementatore della classe di accedere ai precursori di un metodo di qualsiasi livello. Si intende che gli utenti della classe si riferiranno a quello ridefinito.

Si osservi che per ogni classe rimane definito l'insieme dei suoi **discendenti** (**supertipi**) e dei suoi **discendenti** (**sottotipi**).

Molti linguaggi ad oggetti consentono solo l'**ereditarietà semplice**: ogni classe ha al massimo un padre. La classificazione degli oggetti è quindi strettamente gerarchica. Se esiste un'unica radice questa ha nomi come **Object** o **ANY**.

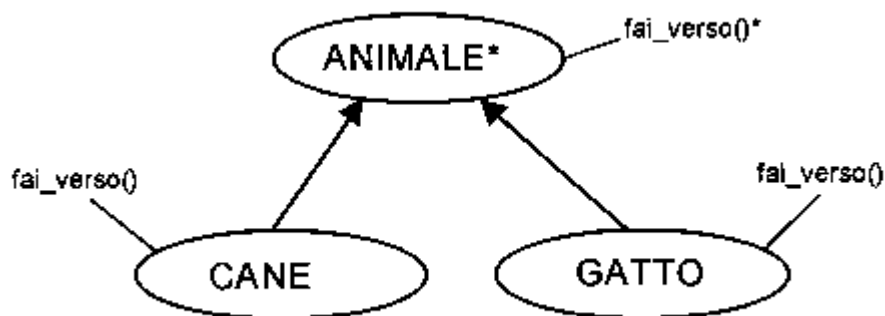
I linguaggi ad oggetti più evoluti consentono l'**ereditarietà multipla**: cioè la possibilità per una classe di ereditare da più di una classe (cioè di avere più padri). Sorgono però problemi non banali di *ereditarietà ripetuta*: esiste cioè la possibilità di ereditare più di un metodo con lo stesso nome e occorre risolvere in qualche modo le ambiguità che si vengono a creare.

Polimorfismo. Tra i vari tipi di polimorfismo che possono essere presenti in un linguaggio, quello che si deve necessariamente ritrovare nella programmazione ad oggetti è noto come **polimorfismo di inclusione universale**, basato proprio sul concetto di ereditarietà. Questa forma di polimorfismo si basa sui seguenti presupposti:

- a) la possibilità di usare *variabili polimorfe*, cioè che possono riferirsi ad oggetti di tipi diversi (generalmente "inclusi" in una certa gerarchia). In pratica basta permettere ad una variabile di tipo T di ricevere un oggetto di un qualsiasi sottotipo di T.
- b) la possibilità di effettuare *chiamate polimorfe*, cioè di indicare con lo stesso nome dei metodi che appartengono a classi diverse e che sono quindi generalmente diversi ("polimorfo" = "avente più forme").

A differenza di altre forme di polimorfismo, come l'**overloading** (sovraccarico o sovrapposizione) degli operatori, il polimorfismo di inclusione universale prevede che *la decisione su quale debba essere la routine da richiamare viene presa a tempo di esecuzione* a seconda della classe effettiva (più stretta) di appartenenza dell'oggetto rispetto a cui viene fatta la chiamata. Questa tecnica è nota come **collegamento dinamico** (late o dynamic binding) dei nomi al codice che deve essere effettivamente eseguito. Esso si contrappone al tradizionale **collegamento statico** (early o static binding) deciso dal compilatore, di norma, nel caso di chiamate non polimorfe.

Spesso il polimorfismo viene introdotto con le **classe astratte**, vale a dire classi in cui compaiono uno o più metodi la cui definizione viene rinviata alle sottoclassi (**metodi astratti**). Tipico è l'esempio rappresentato nella figura qui sotto, in cui la classe ANIMALE ha un metodo astratto `fai_verso()` che trova una concreta realizzazione (definizione) solo nei discendenti "concreti" (ad es. CANE, GATTO, ecc.).



Il polimorfismo trova applicazione soprattutto in schemi simili a quello esemplificato dal seguente frammento:

```
per ogni a : ANIMALE in S esegui
  a.fai_verso()
fineperogni
```

Si immagina che la variabile *a* assuma di volta in volta i vari oggetti (di tipo ANIMALE) che si trovano in un insieme o in un array *S*. Alla variabile *a* verrà quindi di volta in volta associato un animale potenzialmente diverso e a priori sconosciuto. Quindi, per il compilatore, la chiamata *a.fai_verso()* è polimorfa e, come tale, soggetta al collegamento dinamico.

Genericità. Un'altra delle caratteristiche desiderabili dei linguaggi OOP *tipati* è la **genericità**, cioè la possibilità di definire delle classi con uno o più parametri di tipo "tipo". Questa possibilità è utile soprattutto quando si vogliono definire delle classi di tipo **contenitore di dati**: array, pile, code, alberi, ecc.

L'idea è quella di estendere la possibilità di dichiarare il tipo degli elementi presente per il tipo predefinito **array** (ad es. array of integer, array of string, ecc.) ai contenitori definiti dall'utente (pila of integer, pila of string, ecc.).

In assenza di questo strumento (ad es. Java ne è sprovvisto) si è costretti a definire una classe di oggetti "qualsiasi" (ANY o Object) e fare quindi un **downcasting**, cioè una conversione forzata al sottotipo che ci interessa di volta in volta.

Il ricorso al downcasting si rivela comunque necessario in altre occasioni, come quelle che si presentano quando dobbiamo elaborare una collezione di oggetti di varie classi utilizzando le loro operazioni specifiche (non polimorfe).

A volte esiste la necessità di restringere i possibili parametri attuali di tipo "tipo" a quelli che possiedono determinate operazioni (tipico è il caso dell'operazione di confronto : <, >, ecc.). Alcuni linguaggi (ad es. Eiffel e Sather) mettono a disposizione a tale proposito la cosiddetta **genericità vincolata**: solo le classi discendenti di una certa classe specificata possono essere sostituite al posto del parametro.

IV.2 Aspetti avanzati della programmazione a oggetti

IV.2.1 Copia e uguaglianza

La semantica di default per le operazioni di assegnamento e di confronto per uguaglianza fra oggetti prevede che vengano coinvolte solo le identità degli oggetti (in pratica, spesso, dei puntatori di memoria). Nel caso dell'assegnamento questo può creare degli spiacevoli effetti collaterali dovuti alla condivisione di strutture dinamiche che si viene a creare. Per evitarli occorre effettuare l'assegnamento con operazioni del tipo *a:=copia(b)* che ha l'effetto di creare in *a* un puntatore ad una copia dell'oggetto indicato in *b*.

Nel confronto *x = y* tra oggetti, se si vuole che il confronto non venga fatto sulle identità (questo è quasi sempre il caso) ma bensì sullo stato dei due oggetti, è necessario usare operatori diversi come, ad es., **uguale(x,y)**.

In genere i linguaggi mettono a disposizione del programmatore la possibilità di ridefinire **copia** e **uguale** classe per classe.

IV.2.2 Asserzioni

Le **asserzioni** in un linguaggio di programmazione sono delle espressioni booleane valutabili a tempo di esecuzione che possono assolvere a diversi scopi:

- forma primitiva di *specifica* e di *documentazione*
- *prevenzione contro possibili malfunzionamenti del software*; in particolare nella OOP come *precondizioni* e *postcondizioni* dei metodi
- strumento di verifica durante la fase di *testing*
- possibili clausole di uno stile di programmazione noto come *progetto per contratto*

I linguaggi OOP si differenziano notevolmente rispetto al supporto e alla varietà di tipi di asserzioni che mettono a disposizione del programmatore.

IV.2.3 Eccezioni

Per aumentare la *robustezza* di un'applicazione, tutti i più recenti linguaggi OOP mettono a disposizione la possibilità definire, sollevare e trattare **eccezioni**, cioè eventi causati da malfunzionamenti o imprevisti hardware o software che si possono sempre verificare. In particolare il programmatore di ogni metodo ha la possibilità di catturare un'eccezione, riconoscerla e prendere le necessarie azioni di recupero almeno per mantenere l'oggetto in uno stato interno consistente. Il trattamento standard di default è quello di propagare l'eccezione al chiamante, fino eventualmente a raggiungere il metodo avviato per primo, facendolo abortire.

IV.2.4 Contenitori e iteratori

Una volta definito come classe un **contenitore astratto** di dati (ad es. una pila, una lista, una tabella, ecc.) si può presentare il problema di *attraversare* uno di questi contenitori, cioè di costruire cicli del tipo

portati sul **primo elemento**

finché non sei giunto all'ultimo e non hai trovato una condizione di arresto

elabora l'**elemento corrente**

passa all'**elemento successivo**

finefinché

La soluzione più generale, flessibile e astratta a questo problema prevede il ricorso a degli oggetti chiamati **iteratori** la cui classe di appartenenza viene definita in "simbiosi" a quella dei contenitori che li attraversano. In pratica essi assolvono ad un ruolo molto simile a quello rivestito dagli indici dei "for" per l'attraversamento degli array. La differenza è che viene completamente nascosta la rappresentazione dei cursori. Il cursore è l'oggetto che all'interno di un iteratore "punta" all'elemento del contenitore in corso di attraversamento. Questo permette un domani di cambiare sia la rappresentazione dei contenitori che quella degli iteratori senza cambiare il codice delle applicazioni "clienti".

IV.2.5 Distruttori

Oltre ai costruttori può essere talvolta necessario che il programmatore definisca dei metodi, chiamati spesso **distruttori**, che assolvono al compito di svolgere delle azioni finali prima che l'oggetto venga completamente distrutto e le sue aree di memoria rese disponibili per operazioni di allocazione di nuovi oggetti. In particolare, in C++, i distruttori hanno spesso l'onere di recuperare parte della memoria occupata dall'oggetto. Da questo onere sono liberi tutti i linguaggi che prevedono il **garbage collector**, cioè quel processo della macchina virtuale del linguaggio che assolve appunto al compito di individuare e recuperare le aree occupate da oggetti che si rendono inaccessibili ("garbage", appunto).

IV.2.6 Persistenza

La **persistenza** nella OOP è la proprietà di un oggetto di sopravvivere al processo che l'ha creato. Un modo per implementare la persistenza è tipicamente quello di "salvarlo" su un supporto di memoria persistente, come il disco. I linguaggi OOP adottano varie tecniche per fornire questo tipo di servizio all'utenza. Il più semplice e primitivo è quello di mettere a disposizione delle primitive di salvataggio e recupero di un oggetto su un file con un certo nome. Il più sofisticato (ma anche il più semplice da usarsi) è quello di prevedere la possibilità di dichiarare come persistenti alcune classi: gli oggetti creati saranno automaticamente resi persistenti.

IV.2.7 Concorrenza e reti

Fin dal suo concepimento, nel paradigma ad oggetti è stato preso in considerazione l'ipotesi di considerare ogni oggetto come avente un proprio "thread of control". In particolare, Smalltalk vedeva la chiamata di un metodo come la spedizione di un *messaggio* da parte di un *oggetto mittente* (sender) ad *uno ricevente* (receiver) contenente la richiesta di un particolare *servizio* il cui termine veniva comunicato al mittente assieme all'eventuale restituzione di un valore. Con una visione più moderna, si può considerare la chiamata di un metodo come la richiesta di un servizio ad un *oggetto servente* in grado di accettare e ordinare opportunamente le richieste provenienti dai vari oggetti clienti. Alcuni linguaggi ad oggetti mettono a disposizione strumenti od estensioni a supporto della concorrenza e/o della distribuzione degli oggetti.

IV.3 Le classi in C++

Dalle precedenti considerazioni si evince che una classe è una sorta di *modello* che definisce l'aspetto di un *oggetto*. Una classe specifica sia il codice che i dati. Il linguaggio C++ utilizza le specifiche contenute in una classe per costruire degli oggetti. Dunque gli oggetti sono le istanze di una classe. È importante ricordare bene questa distinzione: una classe è un'astrazione logica. Una classe non esiste in memoria finché non si crea un oggetto di tale classe, il quale sarà una rappresentazione fisica della classe in memoria.

Vediamo ora qual è l'aspetto generale di una classe. Per creare una classe si usa la parola riservata **class**. La forma generale della dichiarazione di una classe è la seguente:

```
class nome-classe{  
  funzioni e dati privati  
public:  
  funzioni e dati pubblici  
} elenco oggetti;
```

In questo caso *nome-classe* specifica il nome della classe. Questo nome diviene un nuovo tipo utilizzabile per creare oggetti di tale classe. Una volta dichiarata una classe, diventa possibile creare oggetti di tale classe.

Una classe può contenere membri pubblici e privati. Normalmente tutti gli elementi definiti in una classe sono *privati*. Questo significa che possono accedervi solo i membri della stessa classe, mentre non sarà possibile accedervi dalle altre parti del programma. Questo è il modo in cui si ottiene l'*incapsulazione*: il fatto che i dati siano privati consente di controllarne in modo preciso l'accesso.

Per rendere pubbliche (ovvero accessibili alle altre parti del programma) determinate parti di una classe, occorre dichiararle dopo la parola riservata *public*. Tutte le variabili o funzioni definite dopo lo specificatore *public* saranno accessibili dalle altre parti del programma.

In generale una classe *raggruppa informazioni* che hanno una *stretta relazione* fra di loro.

In definitiva, in C++ una classe crea un nuovo tipo di dati che può essere utilizzato per creare degli oggetti. In particolare una classe crea una struttura logica che definisce una relazione fra i suoi membri. Quando si dichiara una variabile di una classe, si crea un oggetto. Quest'ultimo *occupa spazio in memoria*, mentre la definizione della classe no.

Facciamo ora un esempio. Supponiamo di voler creare una classe che incapsula informazioni sui veicoli: automobili, furgoni e camion.

```
Class veicoli {  
public:  
    char modello[15]; // modello di veicolo  
    int cilindrata; // cilindrata del veicolo  
    int velMax; // velocità max  
};
```

La classe definisce tre variabili d'istanza: *modello*, *cilindrata* e *velMax*. Si noti che *veicoli* non contiene alcuna funzione. Pertanto attualmente è una classe costituita unicamente da dati.

Le variabili d'istanza vengono dichiarate nel seguente modo:

tipo nome-variabile;

Qui *tipo* specifica il tipo della variabile e *nome-variabile* è il nome della variabile. Di conseguenza le variabili d'istanza vengono dichiarate come ogni altra variabile.

La definizione della classe crea un nuovo tipo di dati denominato *veicoli*. Dunque si potranno dichiarare nuovi oggetti di tipo *veicoli*.

Per creare oggetti di tipo *veicoli* occorre utilizzare un'istruzione di dichiarazione come la seguente:

```
veicoli auto; // crea un oggetto veicoli chiamato auto
```

Dopo l'esecuzione dell'istruzione, *auto* sarà un'istanza della classe *veicoli*. Ora la classe avrà una "realtà fisica".

Ogni volta che si crea un'istanza di una classe, si crea un oggetto che contiene una propria copia di ogni variabile d'istanza definita dalla classe. Pertanto ogni oggetto *veicoli* conterrà una propria copia delle variabili d'istanza *modello*, *cilindrata*, *velMax*. Per accedere a queste variabili si utilizza l'operatore punto (.). Quest'ultimo concatena il nome dell'oggetto con il nome di uno dei suoi membri. Ecco il modo in cui si utilizza in generale l'operatore punto:

oggetto.membro

Pertanto, l'oggetto deve essere specificato alla sinistra e il membro alla destra del punto.

Per esempio, per assegnare il valore 200 alla variabile *velMax* di *auto* si usa la seguente istruzione:

```
auto.velMax=200;
```

In generale, l'operatore punto consente di accedere alle variabili d'istanza e di richiamare le funzioni della classe.

La maggior parte delle classi, oltre a contenere dati, contiene anche delle funzioni. In generale le funzioni membro manipolano i dati definiti dalla classe e, in molti casi, offrono l'accesso a tali dati, tipicamente le altre parti del programma interagiranno con la classe tramite le sue funzioni.

Si consideri ad esempio la seguente classe:

```
Class calciatori{  
public:  
    char cognome[15]; //cognome del calciatore  
    char nome[15]; //nome calciatore  
    int gol; //gol segnati  
    int partiteGiocate; //numero di partite giocate  
  
    float mediaGol(); //restituisce la media-gol  
};
```

Come si può notare, la classe denominata *calciatori*, ha quattro membri pubblici di tipo dati e una funzione membro denominata *mediaGol()* anch'essa pubblica.

La dichiarazione della funzione *mediaGol()* avviene nella definizione della classe e quindi non ha bisogno di essere dichiarata ulteriormente.

Per implementare una funzione membro, occorre dire al compilatore la classe cui appartiene la funzione, qualificando il nome della funzione insieme al nome della classe. Nel nostro esempio:

```
//implementa la funzione membro mediaGol()  
float calciatori::mediaGol() {  
    return (gol/partiteGiocate);  
}
```

Si noti che il nome della classe *calciatore* è separato dal nome della funzione *mediaGol* dal simbolo `::`. Il simbolo `::` è chiamato *operatore di risoluzione della visibilità*. In pratica collega il nome di una classe con il nome di un membro, per dire al compilatore la classe cui appartiene il membro.

Il corpo della funzione *mediaGol* è costituito dalla seguente riga:

```
return (gol/partiteGiocate);
```

Questa istruzione restituisce la media-gol di un calciatore in base ai gol realizzati ed al numero di presenze. All'interno di *mediaGol()* è possibile accedere direttamente alle variabili d'istanza *gol* e *partiteGiocate* senza specificare il nome dell'oggetto o l'operatore punto. Quando una funzione membro utilizza una variabile d'istanza che è definita dalla sua classe, lo può fare direttamente, senza fare esplicitamente riferimento a un oggetto e senza utilizzare l'operatore punto.

Una funzione membro deve essere richiamata su un determinato oggetto. In generale una funzione membro può essere richiamata dal codice che si trova all'esterno della sua classe. In questo caso

occorre utilizzare il nome dell'oggetto e l'operatore punto. Per esempio ecco come richiamare *mediaGol()* su un oggetto calciatore denominato *attaccante*:

```
ris=attaccante.mediaGol();
```

Nella variabile *ris* verrà memorizzato il valore ritornato dalla funzione membro *mediaGol()*.

IV.4 Costruttori e distruttori

Negli esempi precedenti alcune delle variabili d'istanza di ciascun oggetto veicoli sono state impostate manualmente utilizzando una sequenza di istruzioni come ad esempio:

```
auto.velMax=200;
```

In generale esiste un modo più semplice per ottenere questo risultato: impiegare un **costruttore**.

Un costruttore inizializza un oggetto al momento della creazione. Un costruttore ha **lo stesso nome della classe** e ha una sintassi simile ad una funzione. Ma al contrario di una funzione, i costruttori non restituiscono alcun tipo esplicito. Ecco dunque la forma generale di un costruttore:

```
nome-classe() {  
    // codice del costruttore  
}
```

Si consideri per esempio la seguente classe:

```
class valori{  
public:  
    int a;  
    valori(); //costruttore  
};
```

Come si può osservare dall'esempio precedente, il costruttore ha lo stesso nome della classe. Nella dichiarazione di una funzione membro all'interno di una classe è presente il tipo di valore ritornato dalla funzione (*int*, *float*,...), nel caso del costruttore tale dichiarazione non c'è. Si è infatti precedentemente affermato che un costruttore non ritorna alcun valore. In definitiva per distinguere un costruttore da una funzione membro, basta osservare due aspetti:

1. il nome del membro è uguale a quello della classe
2. non compare il tipo di dati ritornato nella dichiarazione del membro stesso

Facendo attenzione a queste due proprietà si evita l'errore di confondere un costruttore con una qualunque funzione membro.

L'implementazione (realizzazione) di un costruttore avviene fuori dalla definizione della classe secondo la seguente sintassi:

```
nome-classe::nome-classe() {
    // istruzioni
}
```

Per esempio, la definizione del costruttore valori()

```
valori::valori() {
    a=10;
}
```

mette in evidenza la mancanza dell'istruzione *return*, al contrario della definizione della funzione membro. Inoltre l'istruzione *a=10* fa sì che venga assegnato alla variabile membro *a* il valore 10. Infatti, in genere si utilizza un costruttore per assegnare dei valori iniziali alle variabili d'istanza definite dalla classe. L'interrogativo è: *quando è che avviene l'inizializzazione?*

Si noti che il costruttore dichiarato nella classe *valori* è specificato nell'area *public*. Questo è dovuto al fatto che il costruttore verrà richiamato dal codice che si trova all'esterno della classe. *Il costruttore viene richiamato al momento della creazione di un oggetto*. Per esempio nella riga:

```
valori oggetto1;
```

il costruttore di *valori* viene richiamato sull'oggetto *oggetto1*, assegnando a *oggetto1.a* il valore 10.

Il complemento di un costruttore è chiamato **distruttore**. In molti casi, un oggetto deve eseguire alcune azioni nel momento in cui viene distrutto. Gli oggetti locali vengono creati nel momento in cui si entra nel loro blocco e distrutti quando si esce dal loro blocco. Gli oggetti globali vengono invece distrutti al termine del programma. Vi sono vari motivi per impiegare un distruttore. Per esempio, un oggetto può dover liberare la memoria che era stata allocata precedentemente. In C++ queste operazioni vengono gestite dal distruttore.

Il distruttore ha lo stesso nome del costruttore preceduto però dal simbolo *~*. Anche i distruttori, come i costruttori, non restituiscono alcun valore. La sintassi è la seguente:

```
~nome-classe();
```

Nell'esempio precedente,

```
class valori{
public:
    int a;
    valori(); //costruttore
    ~valori(); //distruttore
};
```

L'implementazione di un distruttore è del tutto simile a quella di un costruttore con la sola variante del carattere *~* prima del nome.