La programmazione Orientata agli Oggetti: OOP

Il modo di *pensare "per categorie"* è tipico dell'uomo: tutti noi tentiamo di organizzare le conoscenze classificandole ed affrontiamo lo studio dei problemi cercando di suddividerli a seconda di una particolare tipologia *astraendo* dal caso particolare gli elementi ritenuti caratteristici ed i comportamenti comuni.

Il paradigma nella programmazione **O**bject **O**riented traduce il pensare per categorie e tende a creare delle "*unità*" indipendenti di codice (gli ***oggetti***) dotate in un certo senso di vita propria, come gli oggetti del mondo reale, che possano essere riutilizzate il più possibile all'interno dello stesso programma o in programmi diversi.

Gli *oggetti* vengono creati partendo da modelli più generali denominati *classi*.

Una ***classe*** è in pratica una famiglia di oggetti, di cui contiene tutte le caratteristiche e i comportamenti comuni.

Le classi sono la descrizione astratta, gli oggetti ne costituiscono le *istanze*, ovvero la realizzazione concreta.

Una classe descrive le "caratteristiche generali" di oggetti e può essere vista come il mattone per implementare algoritmi[1]: una "scatola" che ingloba dati (*attributi* degli oggetti) e funzioni (*metodi*) e classifica con un nome una collezione di oggetti.

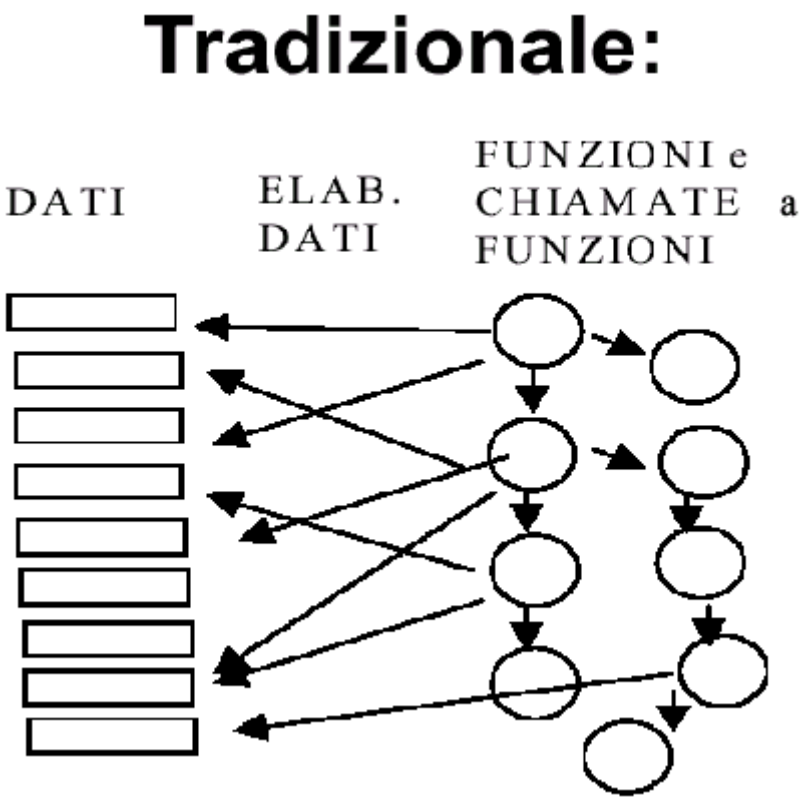
Siamo abituati a classificare col nome (astratto) ad esempio "cane" un animale con 4 zampe (attributo) che abbaia (metodo); a tale classe apparterrà il cane lupo Rintintin (istanza di un oggetto che occuperà memoria).

DESCRIZIONE DI UN SISTEMA A OGGETTI

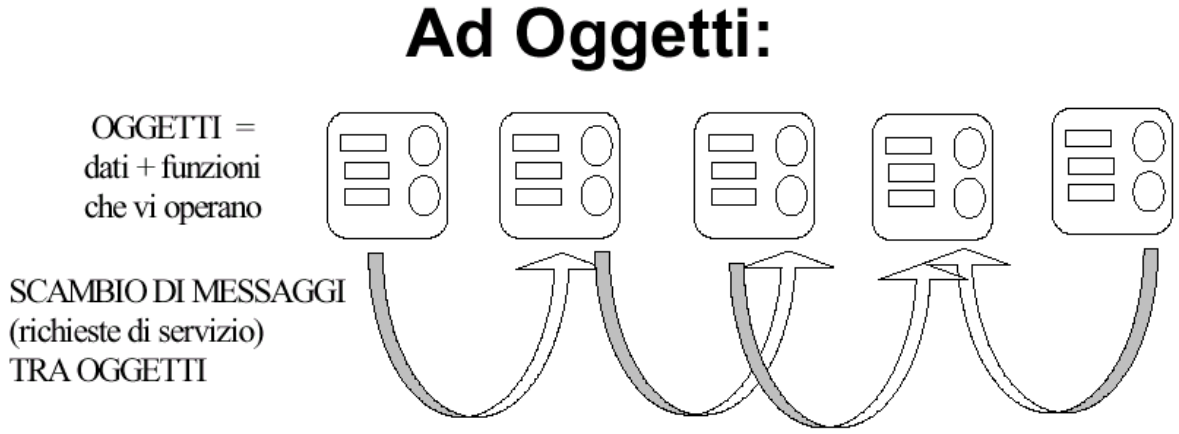
- Ogni oggetto appartiene a ("*è istanza di*") una data ***classe***
- **La classe *racchiude e incapsula la specifica*** di
  - **struttura** dell'oggetto (dati)
  - **comportamento** dell'oggetto (operazioni)

[1] Nel linguaggio Java (OOP "puro") un programma è una classe e classi ed oggetti sono le uniche, o quasi, astrazioni del linguaggio. Ad eccezione dei tipi primitivi di base (int, float, double, char ...) tutto il resto sono oggetti.

In generale, nella *soluzione di un problema* si individuano **dati** e **funzionalità** (azioni) potendola implementare, a livello più semplice, in un programma costituito da istruzioni e dati:



nella progettazione **OO** il *codice che manipola* i dati è invece ***incapsulato*** con la dichiarazione e memorizzazione di quei dati.



Si pensi all'***incapsulamento*** come ad un involucro protettivo che avvolge sia le istruzioni che i dati che si stanno manipolando. Tale involucro definisce il comportamento e *protegge da accessi arbitrari* da parte di un altro programma; il pregio è che ognuno può accedere al codice e *usarlo senza preoccuparsi dei dettagli di implementazione*.

L'essenza della **programmazione Orientata agli Oggetti** è trattare come entità concrete *oggetti* anche astratti con un proprio specifico comportamento che *rispondono a messaggi* che dicono loro di fare qualcosa; una sequenza di passi di un procedimento tipica della programmazione con paradigma "procedurale" può diventare una *collezione di messaggi* tra *oggetti autonomi*.

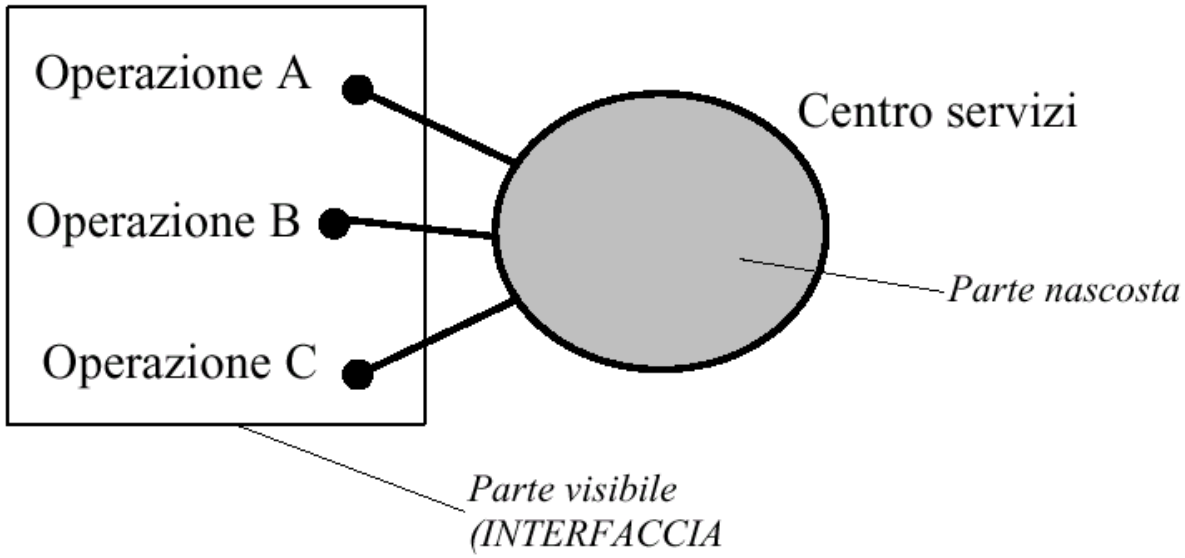
Quindi, all'interno dell'impostazione procedurale, si introduce un ***GROSSO CAMBIAMENTO DI PARADIGMA:***

- **NON** si richiede a funzioni di operare su dati
  nome_funzione (parametro1, parametro2, ….);

- **MA** si richiede alle entità di eseguire delle funzioni
  oggetto.nome_funzione (parametro1, parametro2, ….);

IL CONCETTO DI OGGETTO

- Un oggetto come ***astrazione di dato*** o ***centro di servizi***
- con una *parte visibile* → *INTERFACCIA*
- e una *parte nascosta*



LE OPERAZIONI DI *INTERFACCIA* RAPPRESENTANO:

- I servizi che gli altri oggetti possono richiedere all'oggetto
- Modalità di trattamento dei dati dell'oggetto o di accesso ai dati

LA PARTE NASCOSTA CONTIENE:

- **DATI**
  • Gli *attributi* che caratterizzano l'oggetto
  • Lo *stato* dell'oggetto
  → Sono accessibili solo attraverso richieste di servizio

- **OPERAZIONI PRIVATE**
  • Sfruttate dalle operazioni pubbliche ma non servizi esterni

MODELLO CLIENTE/SERVITORE con spostamento del "fuoco"

- **NON** chiamare una funzione che elabori dei dati

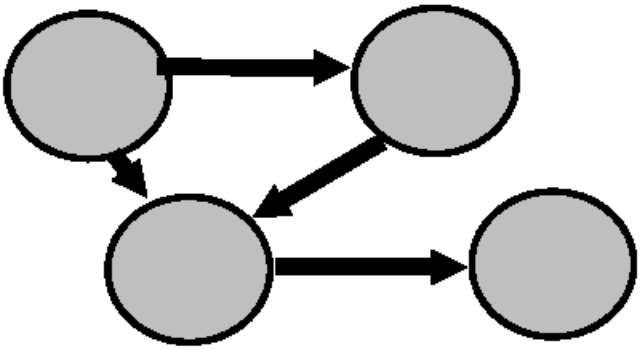
- **MA** chiedere a una entità software di svolgere un servizio

*Esempi*o: dato un file, NON invocare una funzione che scriva dati sul file,
MA chiedere al file stesso di svolgere il servizio relativo alla scrittura di file su di sé

- I clienti *non conoscono* l'organizzazione interna dei centri di servizio, e *non possono* accedere *direttamente* a essa
- non interessa l'algoritmo (come sono fatte le cose) ma interessa che le cose vengano fatte
- i clienti sono indipendenti da come sono fatti i servitori: facilita la manutenzione!

ARCHITETTURA DI UN SISTEMA A OGGETTI

- Un ***insieme di oggetti***
- che **intergiscono gli uni con gli altri**
- senza conoscere nulla delle rispettive rappresentazioni concrete



Gli altri meccanismi che rinforzano il modello **object-oriented**, oltre l'*incapsulamento*, sono: l'*ereditarietà* e il *polimorfismo*.

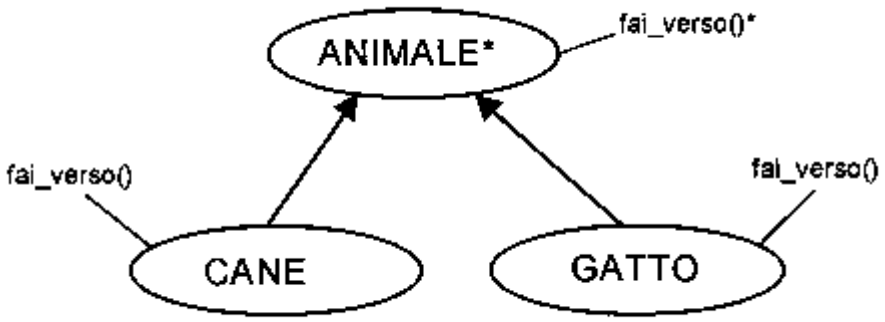
EREDITARIETÀ

Ogni classe derivata eredita tutte le proprietà della classe base di partenza.

Le classi che ereditano metodi o attributi da un'altra vengono dette *sottoclassi* di quest'ultima che è chiamata *superclasse*.

POLIMORFISMO

Diverse classi possono svolgere le stesse azioni in modo differente. Ad esempio il metodo fai_verso() per un cane sarà abbaiare, per un gatto miagolare.



VANTAGGI DELLA PROGRAMMAZIONE AD OGGETTI

- Facilitazione di costruzione *cooperativa* di software:
  - diverse persone sviluppano diverse classi
  - ogni programmatore può semplicemente verificare il comportamento delle sue classi istanziandone oggetti e verificandone il comportamento in risposta a richieste di servizio
  - unico accordo necessario per integrare il tutto in un unico sistema finale: definire le ***interfacce*** delle classi

- Facilitazione della *gestione e manutenzione*
  - Se vi sono errori sui dati in un oggetto, è facile scoprire dove si trova l'errore, perché (siccome i dati non sono visibili all'esterno dell'oggetto) esso non potrà essere che all'interno dell'oggetto che gestisce quei dati
  - le modifiche a una classe non rendono necessario modificare il resto del programma (le altre classi) a meno che non venga modificata l'interfaccia

- Supporto a progettazione e sviluppo *incrementali*: si possono definire nuove classi sfruttando il codice di classi già esistenti

- Possibilità di *rapida prototipazione*: non importa che una classe sia completamente definita per poter iniziare a "testare" il funzionamento degli oggetti di quella classe.

Linguaggio C++

Nel linguaggio C++ (inizialmente chiamato “C con classi”, esempio di linguaggio ad Oggetti "non puro") si aggiunge l'astrazione di classe e oggetto ai costrutti tradizionali della programmazione; con *class* si definisce, infatti, un *particolare tipo aggregato*: un tipo definito da utente.

Flussi di I/O come oggetti

Input e output si basano sull’uso di *flussi* cioè sequenze di byte che viaggiano da un’origine ad una destinazione lungo un percorso di comunicazione.

Se scrive su un flusso, il programma è l’*origine* del flusso stesso; se legge da un flusso è la *destinazione* del flusso. Il percorso di comunicazione dipende dal tipo di I/O effettuato: può consistere in un trasferimento da memoria a memoria, in un file system, in una rete o in altre forme di I/O.

Si ricordino, ad esempio, gli oggetti "***collegati***" ai ***dispositivi standard*** di **I/O** (i tre **principali**: **stdout, stderr**, **stdin**) che consentono letture e scritture che non avvengono direttamente ma tramite un'area di memoria, detta ***buffer***.

Quindi:

cout è l’identificatore di un oggetto di tipo “*stream di output*” il cui ***costruttore*** è definito nella Libreria Standard (iostream); si userà per scrivere a video con sintassi

cout << variabile; oppure cout <<”*stringa”;*

Come pure:

cin è l’ identificatore di un oggetto di tipo “stream di input” il cui ***costruttore***[2] è definito nella Libreria Standard (iostream); si userà per leggere da tastiera con sintassi:

cin>> variabile;

Operatori di flusso di I/O

In **C++** sono infatti definiti gli **operatori** di **flusso** di **I/O** : **<<** (**inserimento**)
>> (**estrazione**)

In una stessa istruzione si possono "***impilare***" più operazioni di **inserimento** una di seguito all'altra.

Esempio: **cout << dato1 << dato2 << dato3;**
equivale a: **cout << dato1; cout << dato2; cout << dato3;**

Come le operazioni di **inserimento**, anche quelle di **estrazione** possono essere "***impilate***" una di seguito all'altra in un'unica istruzione.

Esempio: **cin >> dato1 >> dato2 >> dato3;** (i dati **dato1**, **dato2**, **dato3** devono essere forniti nello stesso ordine).

Il programma interpreta la lettura di un dato come **terminata** se incontra un **blank**, un **carattere di tabulazione** o un **ritorno a capo**.

---

[2] *Il **costruttore** è un metodo particolare usato per inizializzare un nuovo oggetto* che ha lo stesso nome della classe.

Creare un'istanza di una classe

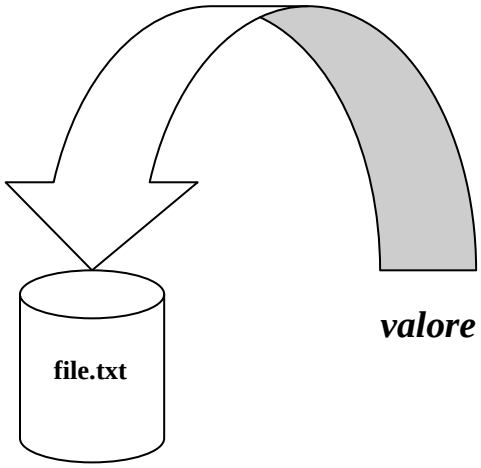
E' anche possibile **creare un oggetto** personalizzato di tipo "*stream di output **a file***" ricorrendo alla classe ***ofstream*** il cui ***costruttore*** è definito nelle librerie standard (***fstream***) ricorrendo alla seguente sintassi che permette di creare l'istanza di una classe:

nome_ costruttore_ classe nome_oggetto (parametri);

Pertanto, con la sintassi:

ofstream *nome_flusso* (*"nome_file.estensione"*);

si chiama il metodo costruttore ***ofstream***, passandogli come parametro il nome del ***file di testo*** e si sceglie a piacere il nome del flusso che permetterà di memorizzare nella stessa cartella di lavoro.



Per scrivere sul ***file di testo*** si utilizzerà l'operatore di inserimento con sintassi nota:

nome_flusso **<< variabile**; oppure ***nome_flusso <<"stringa";***

Con la sintassi:

ofstream *nome_flusso* (*" percorso/nome_file.estensione"*);

si potrà memorizzare su ***file di teso*** eventualmente in altra cartella di lavoro (vedremo in seguito come salvare su file di *tipo binario*)

Analogamente come istanza della classe ***ifstream*** si potrà **creare un oggetto** personalizzato di tipo "*stream di input **da file***" con ***costruttore*** definito nella stessa libreria standard (***fstream***) ricorrendo alla seguente sintassi:

ifstream *nome_flusso* (*"nome_file.estensione"*);

In modo più generale si potrà ricorrere alla **classe fstream** (con ***costruttore*** definito nella stessa libreria standard ***fstream***) che serve sia per le operazioni di **input** sia per quelle di **output** ed è particolarmente **indicata per operare su files "*binari*" ad accesso "*casuale*".**

Generalmente, infatti, gli **oggetti** della classe **ifstream** (che serve per le operazioni di **input**) sono associati a files di sola lettura , che possono essere sia in modo "testo" che in modo "binario", ad accesso generalmente "*sequenziale*" come quelli della classe **ofstream** (che serve per le operazioni di **output**) generalmente associati a files di sola scrittura , che possono essere sia in modo "testo" che in modo "binario", ad accesso generalmente "*sequenziale*".

Per costruire un oggetto di **fstream** e associarlo a un *file* si può usare il **costruttore**[3] (che ***crea*** l'oggetto e contemporaneamente ***apre il file*** con le modalità impostate) con la seguente sintassi:

fstream *nome_flusso* ("*nome_file.estensione*", *modo*);

- Il *nome del file* viene sempre passato al costruttore come **stringa**

- Il secondo argomento è un **enumeratore** (della classe **ios_base**) o una combinazione di **enumeratori** (tramite OR bit a bit), indicanti il "*modo*" di *apertura* del file con eventuale *protezione*

Principali "*modi*" di apertura di un file:

ios_base::app	Apre un file di **output** e **si posiziona permanentemente in fondo**
ios_base::ate	Apre un file **qualsiasi** e **si posiziona inizialmente in fondo** "**a**t **t**he **e**nd"
ios_base::binary (*)	Apre un file in modo "**binario**"
ios_base::in	Apre un file di **input**
ios_base::out (**)	Apre un file di **output**
ios_base::trunc (**)	Apre un file e ne cancella il contenuto

(*) se **ios::binary** non é specificato, il file é aperto di *default* in modo "testo": in questo caso, in output, ogni carattere **NL** é trasformato nella coppia di caratteri **CR**, **LF** (e viceversa in input)

(**) Non si possono aprire oggetti stream con il solo enumeratore ios ::out che deve essere combinato mediante *bitwise* OR ad esempio con ios:: app oppure ios::in
Se si usa su un ofstream senza specificare ios:: app, ios::ate o ios::in implica l'uso implicito di ios::trunc

Esempio:

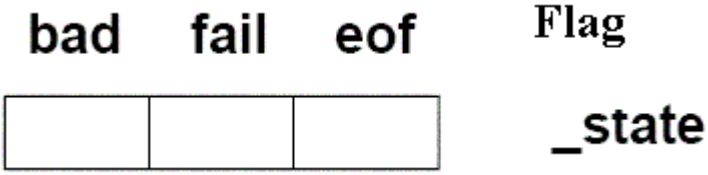
fstream out ("dati.dat", (**ios_base::out**|**ios_base::app**) | **ios_base::binary**);

// **crea** il flusso logico ***out***
// in contemporanea **apre** con **modalità** impostate
// il **file** di tipo **binario** con nome ***"dati.dat"***

[3]Tale costruttore in **overload** ha gli stessi argomenti e svolge la stessa funzione del metodo **open**. Di solito si lascia poi al ***distruttore*** il compito di chiudere il file oltre a svuotare il buffer del file (metodo invocato automaticamente al termine del programma nel caso di oggetti globali o all'uscita del programma dall'ambito di definizione).

In caso di errore, il file non viene aperto; non appaiono messaggi, ma lo "***stato***" dell'operazione può essere controllato mediante apposite *funzioni-membro* (comuni a tutte le classi in quanto derivate da **ios**). In particolare i metodi *good*(), *fail*(), *bad*(), *eof*() permettono di capire se l'ultima operazione effettuata sullo stream rispettivamente è andata a buon fine, ha riportato un errore, l'errore è irrecuperabile, o si è raggiunta la marcatura di fine stream.

In **lettura**, se il risultato dell'**operazione** è **NULL** (e quindi **false**, se convertito in **tipo bool**), vuol dire di solito che si è raggiunta la ***fine*** del ***file*** (***eof***).



Un concetto importante è quello della **"posizione corrente"** in un **oggetto stream** (***file position indicator***), che coincide con l'**indice** (paragonando lo **stream** a un **array**) del prossimo ***byte*** che deve essere **letto** o **scritto**. Ogni **operazione** di **I/O** modifica la **posizione corrente**, la quale può essere anche ricavata o impostata direttamente usando particolari **metodi**. A questo proposito precisiamo che la parola "**inserimento**", usata come sinonimo di **operazione** di **scrittura**, ha diverso significato in base al valore della **posizione corrente**: se questa è interna allo **stream**, i dati non vengono "inseriti", ma sovrascritti; se invece la **posizione corrente** è alla ***fine*** dello **stream** (cioè una posizione oltre l'ultimo ***byte***), i nuovi dati vengono effettivamente inseriti.

Precisiamo che la stessa **posizione corrente** "si muove" in realtà sul **buffer** e non direttamente sull'**oggetto** (anche se il valore assoluto della **posizione** è riferito all'inizio dell'**oggetto**), e quindi alcuni **metodi** di gestione della **posizione corrente** (**tellp** che ricava la **posizione corrente** , **seekp** che imposta la **posizione corrente**, **tellg** e **seekg** analoghi in input), operano effettivamente sul **buffer**. Inoltre, tutte le volte che si parla di **caratteri** "rimossi" da un **oggetto stream** (per esempio con il metodo **ignore**), in realtà si intende dire che sono "rimossi" dal **buffer**, non fisicamente dall'**oggetto**.

In realtà, per costruire un oggetto di **fstream** e associarlo a un *file*, si può operare anche in altro modo, con chiamata esplicita alla *funzione membro* **open()** :

- si dichiara l'**oggetto stream** con un **costruttore** senza **argomenti**
- si chiama la funzione-membro **open**, che "apre" il file, crea l'oggetto di **filebuf** corrispondente e lo "attacca" all'oggetto **stream** precedentemente creato;

(si ricordi che ogni **oggetto** di una classe derivata da **ios** é *attached* (associato) ad un **oggetto** derivato da **streambuf** e le due classi lavorano insieme, la prima per le operazioni di I/O ad alto livello come la *formattazione*, la seconda per l'accesso al *buffer* di I/O e in generale per l'I/O di basso livello)

Conviene ricorrere a tale modalità solo quando si vuole usare lo stesso **oggetto stream** per diversi files sequenzialmente (la funzione **close()** "stacca" il file ma non **distrugge** l'**oggetto stream**).

Esempio: fstream *nome_stream*; // **crea** l'oggetto senza aprire il file

nome_stream.**open**("*nome_file.dat*", ios_base::out|ios_base::app); // **apre** il file

La *funzione-membro* **close()** "*chiude*" il file (scaricando il buffer). E' ***obbligatoria*** quando il file é stato aperto dalla **open()**; se invece il file é stato aperto dal **costruttore**, lo stesso é automaticamente chiuso dal **distruttore**.

Per I/O in ***modo binario***, invece di usare gli operatori di estrazione ed inserzione (<< e >>), sono utili[4] le *funzione-membro*: **write**() per scrivere e **read**(), **readsome**( ) per leggere. I prototipi di tali metodi sono sotto illustrati:

- ostream& **write**(const char* buf, streamsize n); // *funzione-membro* della classe *ostream*
- istream& **read**(char* buf, streamsize n); // *funzione-membro* della classe *istream*
- streamsize **readsom**e(char* buf, streamsize n);

Il tipo **streamsize** (definito nel namespace std) è sinonimo di *int* ed indica un numero di bytes consecutivi in un oggetto stream. E' uno dei **tipi definiti nella Libreria Standard**.

Esempio: **lettura/scrittura da file** di tipo **binario**

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main ()  {
                        ifstream ingr;
                        ingr.open("source.dat", ios_base::in | ios_base::binary);
                        ofstream usc;
                        usc.open("dest.dat", ios_base::out | ios_base::app |  ios_base::binary);

                        char app [4096];
                        int letti = ingr.readsome(app, 4096);    // ritorna il numero di bytes
                                                                 // effettivamente letti
                        while (letti>0) {
                                              usc.write(app, letti);
                                              letti = ingr.readsome(app, 4096);
                         }
                        ingr.close();
                        usc.close();

                        system("PAUSE");
                        return 0;
}
```

NB: sono accettate per retrocompatibilità (ad eccezione dei compilatori più attuali) anche le combinazioni di enumeratori:

ios::in | ios::binary e **(ios::out|ios::app) | ios::binary**

ricordando che la **classe base** della ***gerarchia***, **ios_base**, contiene **proprietà** e **metodi** che sono comuni sia alle **operazioni** di ***input*** che a quelle di ***output*** ed è una classe che descrive in modo *generico* senza dipendenza dai tipi dei ***parametri*** (di **template**) e le stesse caratteristiche sono presenti nella sua **classe derivata**, **ios** (con la differenza che questa è una **specializzazione** con ***argomento* char**)

[4] Con uso dei metodi **get(***char***)** e **put(***char***)** si possono rispettivamente leggere e scrivere file **binari** un *byte* alla volta.