Il linguaggio C++

Principi di OOP nel linguaggio C++

Autore: Prof. Rio Chierego

Marzo 2016

Sommario

- **Linguaggio C++: breve storia**
- **Peculiarità del linguaggio C++ rispetto al C**
  - **iostream.h**
  - il tipo **bool**
  - operatore di **scope resolution ::**
  - qualificatore **const**
  - **linkage**
  - **namespace**
  - direttiva **using**
  - funzioni con **argomenti di default**
  - **overloading** funzioni
  - **reference**
  - la classe **string**
- **Linguaggio C++ ed applicazione dei principi base dell'OOP programming**
- **Esercizi da svolgere**

Linguaggio C++: breve storia

Lo sviluppo del linguaggio C++ all'inizio degli anni Ottanta è dovuto a **Bjarne Stroustrup** dei laboratori

Il linguaggio C++ venne utilizzato all'esterno del gruppo di sviluppo di Stroustrup nel **1983** e, fino all'estate del **1987**, il linguaggio fu soggetto a una naturale evoluzione

Uno degli **scopi principali** del C++ era quello di mantenere piena compatibilità con il C. L'idea era quella di conservare l'integrità di molte librerie C e l'uso degli strumenti sviluppati per il C

Il C++ consente lo sviluppo di software su larga scala. Grazie a un maggiore rigore sul controllo dei tipi, molti degli effetti collaterali tipici del C, divengono impossibili in C++

Il miglioramento più significativo del linguaggio C++ è il supporto della programmazione orientata agli oggetti (**Object Oriented Programming**: OOP), Per sfruttare tutti i benefici introdotti dal C++ occorre cambiare approccio nella soluzione dei problemi. Ad esempio, occorre identificare gli oggetti e le operazioni ad essi associate e costruire tutte le classi e le sottoclassi necessarie.

Linguaggio C++: peculiarità – iostream.h

Stream standard di input e di output: cin e cout

L'header file **iostream.h** contiene la definizione di due stream di I/O – **cout** e **cin** – che consentono di interagire con lo *standard output* (video) e lo *standard input* (tastiera) in modo molto semplice

- **cout** serve per inviare messaggi e dati sull'uscita (video)

Esempio

```
#include <iostream.h>
void main ( ) {
cout << "Il linguaggio C++" << endl;
 ....
cout << "Il valore di a e': " << a;
 }
```

dove **<<** prende il nome di **operatore di inserzione** ("scrivi su")

- **cin** serve per leggere i dati dall'input standard (tastiera)

Esempio

```
#include <iostream.h>
void main ( ) {
float temp;
cout << "Temperatura in gradi C°"<< endl;
cin >> temp;
cout << "Temperatura in gradi F°"<<  32 + t emp * 9/5 << endl;
 }
```

dove **>>** prende il nome di **operatore di estrazione** ("leggi da")

Linguaggio C++: peculiarità –il tipo semplice bool

Il tipo booleano

Per rappresentare il valore di una espressione logica nello standard C++ è stato introdotto un nuovo tipo di dati semplice **bool** che può assumere solo due valori: **true** e **false**

Tipi struct, union ed enum

In C++ ogni definizione **struct**, **union** o **enum** diventa essa stessa un tipo senza usare la parola chiave typedef ed il loro nome(tag) è utilizzabile anche da solo per dichiarare variabili dello stesso genere

Esempio:
```
struct
 {
  char Cognome[30];
  char Nome[30];
  } Cittadino;
Cittadino citt;        //definizione di una variabile di tipo "Cittadino"
```

Linguaggio C++: peculiarità – operatore ::

Operatore di scope resolution ::

Se una variabile locale ha lo stesso nome di una variabile globale all'interno di quel blocco in C non è possibile accedere all'omonima variabile globale (**information hiding**)

In C++ ciò è stato risolto introducendo l'operatore **::** detto di **scope resolution**

Anteposto al nome di una variabile informa il compilatore che si sta facendo esplicito riferimento ad una **variabile globale**

Esempio:

```
#include <iostream.h>
int x = 0;          //variabile globale inizializzata a 0
void main ( )
{
int x = 5;          //variabile locale omomina inizializzata a 5
::x = 4;            // Assegna alla variabile globale omonima il valore 4
cout <<  x << endl;        //verrà mostrato a video il valore 5
cout << ::x << endl;       //verrà mostrato a video il valore 4
}
```

N.B. L'operatore di scope resolution :: permette di accedere solo alla variabile globale omonima

Linguaggio C++: peculiarità – qualificatore const

Qualificatore const

Il qualificatore const consente di dichiarare un oggetto come "non modificabile" eccetto che al momento della sua inizializzazione

In C++ una variabile const è utilizzabile dovunque sia consentito l'uso di una espressione costante

Il segmento di programma C++ che segue è perfettamente lecito

```
const unsigned DIMENSIONE = 100;    //definizione di una variabile const inizializzata al valore 100
 int vettore[DIMENSIONE];           //definizione di una vettore di 100 interi
```

N.B. In C il compilatore darebbe una segnalazione di errore perché non è possibile l'uso di una variabile (anche se const) come dimensione di un vettore statico

Linguaggio C++: peculiarità – linkage

Linkage

In C++ (ma anche in C) l'unità di compilazione è il file ma un programma può essere costituito da numerosi file sorgenti compilati separatamente e collegati insieme dal linker

Il **linkage** determina la porzione di programma nel quale un identificatore può essere referenziato ossia stabilisce il suo **scope** (visibilità)

Se un identificatore è visibile in tutto il file sorgente in cui è dichiarato e la sua dichiarazione contiene lo specificatore di memorizzazione **static** si dice che ha **linkage interno** ossia **è visibile** solo all'interno di quel file sorgente ma non altrove

Se un identificatore è visibile in tutto il file sorgente in cui è dichiarato **ma non e' dichiarato come static** si dice che ha **linkage esterno** ossia **è visibile** da quel punto in tutto il resto del programma

Se la dichiarazione di un identificatore all'interno di un blocco **non contiene** lo specificatore di classe di memorizzazione **extern** si dice che quell'identificatore non ha **nessun linkage** ed è **visibile** solo all'interno di quel blocco

N.B. se si vuole implementare una libreria di **funzioni esportabili** (ossia richiamabili da ogni punto del programma) dovranno avere **linkage esterno** mentre quelle realizzate per scopi interni (quindi non esportabili) **linkage interno**

Linguaggio C++: peculiarità - namespace

Lospazio dei nome: namespace

Tutti gli identificatori con **linkage esterno** condividono un'area di memoria definita **spazio o ambiente globale**

In progetti complessi a cui lavorano molte persone che elaborano numerosi file si potrebbero definire involontariamente identificatori con lo stesso nome **nello spazio o ambiente globale** (evento meno improbabile di quanto si pensi)

Tali eventualità creano al momento di richiamare il linker **conflitti di linkage** non facilmente risolvibili ed evitabili solo con un controllo minuzioso dei programmi già elaborati o con una maggiore attenzione in fase di definizione delle specifiche di progetto

Esempio

```
//Header file name1.h
int x = 2;
…
//Header file name2.h
 int x = 3;
 …
 //Source file main.cpp
#include "name1.h"
#include "name2.h"
….
```

Al momento del linking dei tre file sarà segnalato dal linker un errore del tipo

'x' : redefinition; multiple inizialization

Linguaggio C++: peculiarità - namespace

Il C++ standard offre l'opportunità di **suddividere lo spazio o ambiente globale** in più parti, ognuna delle quali è definita come *"spazio dei nomi"* o **namespace**

In questo modo possono convivere identificatori con lo stesso nome purchè definiti in namespace differenti

La sintassi di un **namespace** è la seguente

namespace *<identificatore>* **{** *<corpo del namespace>* **}**

dove

<identificatore> specifica il nome del namespace

<corpo del namespace> è una lista di dichiarazioni di dati e funzioni

Esempio

```
//Header file name1.h
namespace primo { int x = 2; }

//Header file name2.h
 namespace secondo { int x = 3; }

 //Source file main.cpp
#include <iostream.h>
#include "name1.h"
#include "name2.h"
void main ( )
{
cout << "Somma = " << primo::x + secondo:: x << endl;
}
```

Linguaggio C++: peculiarità – namespace standard std

Tutti gli identificatori del C++ standard sono inseriti entro " lo spazio dei nomi" chiamato std. Per questi motivi un programma scritto in C++ standard assume una forma leggermente differente da quella oramai nota

```
#include <iostream>
void main ( )
{
std::cout << "Il linguaggio C++ standard" << std::endl;
}
```

manca il **.h** per consentire la convivenza con i vecchi header file del linguaggio C

N.B. L'oggetto **cout** ed il manipolatore **endl** sono preceduti dal prefisso **std** che qualifica a quale namespace appartengono

Linguaggio C++: peculiarità – la dichiarazione/direttiva using

La dichiarazione using consente di riferirsi ad un certo identificatore tratto da un namespace semplicemente mediante il suo nome

```
#include <iostream>
using std::cout;
using std::cin;
void main ( )
{
cout << "Il linguaggio C++ standard" << endl;
}
```

Per programmi costituiti da un unico file sorgente o nel caso di identificatori di uso frequente o quando si è sicuri che non si possono creare conflitti, risulterebbe molto più comodo usare direttamente gli identificatori come se appartenessero ad un unico namespace eliminando la necessità di qualificazione (::)

La direttiva using può fornire una soluzione al problema: inserita prima di un identificatore o di un namespace ne modifica lo scope(visibilità) permettendo di importare tutti gli identificatori del namespace std nello spazio o ambiente globale

```
#include <iostream>
using namespace std;
void main ( )
{
cout << "Il linguaggio C++ standard" << endl;
}
```

ATTENZIONE: se se ne abusa si vanifica la funzione dei namespace rendendo possibili di nuovo i conflitti di nome per gli identificatori

Linguaggio C++: peculiarità – argomenti di default per le funzioni

In C++ è possibile attribuire dei **valori di default** ai parametri formali di una funzione

Al momento della chiamata, se non vengono passati gli argomenti corrispondenti, la funzione assumerà come tali quelli di default (ciò è impossibile nel linguaggio C)

Esempio: consideriamo la dichiarazione della seguente funzione

void **ordine** (int primo = **8**, float secondo = **7.1**);

La chiamata della funzione può quindi assumere differenti forme, tutte ugualmente lecite

ordine ( ); // il compilatore assumerà la chiamata **ordine (8, 7.1);**

ordine (**2**) // il compilatore assumerà la chiamata **ordine (2, 7.1);**

ordine (**21, 56.1**); // il compilatore assumerà la chiamata **ordine (21, 56.1);**
// N.B. questa forma è perfettamente equivalente al C

Fare attenzione:

1) Nella lista dei parametri quelli che hanno valori di default devono essere elencati per ultimi nella dichiarazione di una funzione ed uno di seguito all'altro
2) Un argomento di default non può essere ridefinito in una dichiarazione successiva anche se la ridefinizione è identica all'originale (tra prototipo e definizione di una funzione gli argomenti di default devono essere specificati una volta sola)

Linguaggio C++: peculiarità – overloading delle funzioni

In C++ permette la possibilità **dell'overloading delle funzioni**, comprese quelle della libreria standard, ossia di definire funzioni con lo stesso nome ma con una lista di parametri differente (diversa segnatura)

Il compilatore in fase di run-time in base al tipo ed al numero dei loro argomenti è in grado di distinguere quale delle funzioni chiamare (**binding dinamico**)

Non è possibile scrivere funzioni che differiscano unicamente per il tipo del valore di ritorno perchè in questo caso il compilatore non potrebbe discernere quale funzione chiamare

Esercizio: Eseguire in minimo fra due numeri indipendentemente dal loro tipo utilizzando l'overloading

int **minimo** (int x, int y);

float **minimo** (float x, float y);

Linguaggio C++: polimorfismo – l'OVERLOAD (esempio)

```
#include <iostream>

using namespace std;

// Definizione dei prototipi che realizzano l'OVERLOAD della funzione "moltiplica"
int moltiplica(int x, int y);
float moltiplica (float x, float y);


int main(int argc, char** argv)
{
int a = 5;
int b = 6;
int risultato;

float c = 7.9;
float d = 9.2;
float risultato1;

risultato = moltiplica(a, b);
cout << "Il risultato della moltiplicazione tra interi e': " << risultato << endl;

risultato1 = moltiplica (c, d);
cout << "Il risultato della moltiplicazione tra float è: " << risultato1 <<endl;

return (0);
}
```

Linguaggio C++: polimorfismo – l'OVERLOAD (esempio)

```
int moltiplica(int x, int y)
// Dichiarazione della funzione
{
int ris;

ris = x * y;
return ris;
// Valore restituito dalla funzione.
}


float moltiplica(float x, float y)
// Dichiarazione della funzione
{
float ris;

ris = x * y;
return ris;
// Valore restituito dalla funzione.
}
```

Se si prova ad eseguire il precedente esempio, ci si accorgerà che il programma chiamerà in modo automatico la funzione appropriata in base al tipo di argomenti fornito.
Si noti che per poter fare l'overloading di una funzione non basta che soltanto il tipo restituito dalla funzione sia differente, ma occorre che anche gli argomenti lo siano.

Linguaggio C++: peculiarità – reference

La stragrande maggioranza dei linguaggi di programmazione di alto livello (linguaggio C e C++ compresi) dispone di due meccanismi fondamentali per il passaggio di argomenti ad una funzione o procedura:

- **per valore**: la funzione riceve una copia degli argomenti
- **per riferimento**: la funzione riceve una copia dell'indirizzo degli argomenti

Il C++ consente di superare questa dicotomia obbligata tipica del c introducendo un nuovo passaggio dei parametri per riferimento: **il reference**

Un reference è sempre un contenitore di indirizzi, ma si può adoperare come una normale variabile

Il nuovo costrutto richiede l'uso dell'operatore **& (ampersand)** che non ha più il significato di **"indirizzo di"** ma quello di **"riferimento a"**

Il **reference** non è una copia della variabile, **ma la stessa variabile sotto un altro nome (alias)** ovvero un sinonimo

Linguaggio C++: peculiarità – reference

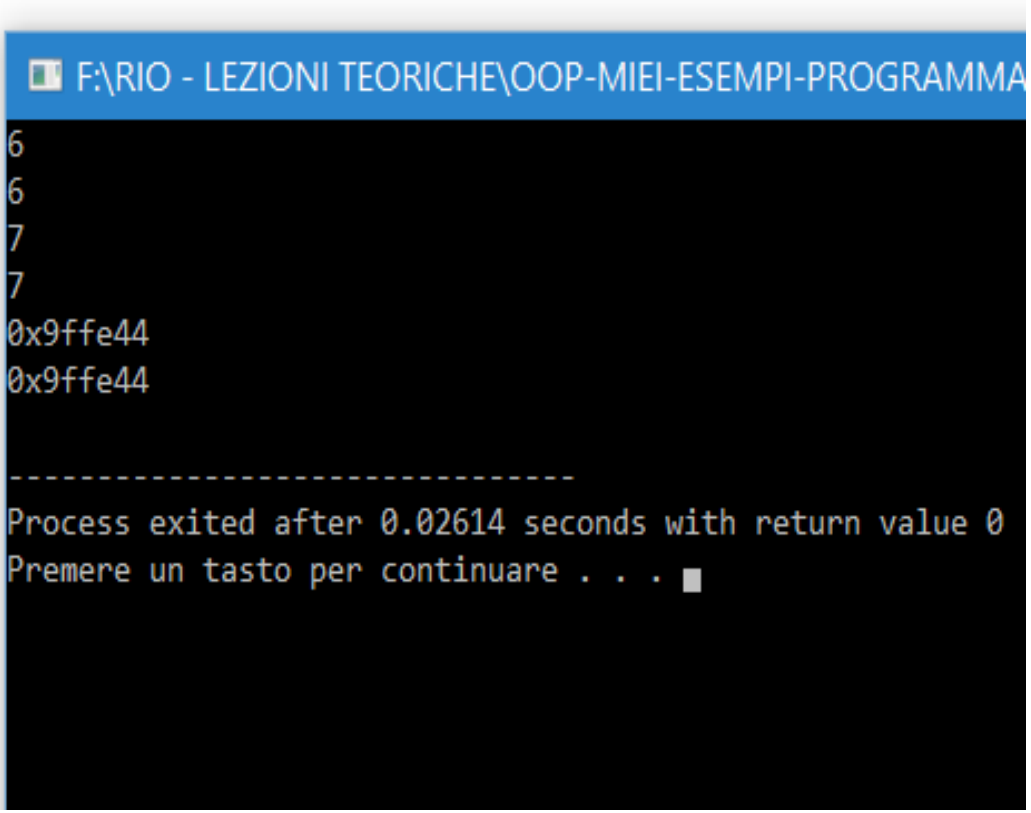
```
#include <iostream>
using namespace std;
void main( )
{
int dato = 5;
int& refdato = dato;

cout << ++dato << endl;
cout << refdato << endl;
cout << ++refdato << endl;
cout << dato << endl;

cout << &dato << endl;
cout << &refdato<< endl;

return;
}
```

L'output prodotto è il seguente:



Linguaggio C++: peculiarità – reference

Supponiamo di voler creare due funzioni di nome **minimo** in grado di determinare il valore più piccolo tra due interi.

```
#include <iostream>

using namespace std;

void minimo (int, int, int*);
void minimo (int, int, int&);

int main(int argc, char** argv)
{
int x, y;
int m, m1;

cout << "Primo numero intero : ";
cin >> x;
cout << "Secondo numero intero : ";
cin >> y;

cout << endl << "Funzione minimo con passaggio parametro x INDIRIZZO";
minimo (x, y, &m);   //terzo parametro passato per indirizzo
cout << endl << "Il minimo tra " << x << " e " << y << " e' = " << m;

cout << endl << endl << "Funzione minimo con passaggio parametro x REFERENCE";
minimo (x, y, m1);   //terzo parametro passato per reference
cout << endl << "Il minimo tra " << x << " e " << y << " e' = " << m1;

return (0);
}
```

Overloading

Chiamata alla funzione minimo con parametro passato per indirizzo

Chiamata alla funzione minimo con parametro passato per reference

Linguaggio C++: peculiarità – reference

```
// Funzione minimo passaggio parametro x INDIRIZZO
void minimo (int a, int b, int* c)
{
if(a <= b)
    {
    *c = a;
    }
else
    {
    *c = b;
    }

return;
}
```

Funzione minimo con parametro di tipo puntatore

```
// Funzione minimo passaggio parametro x REFERENCE
void minimo (int a, int b, int& c)
{
if(a <= b)
    {
    c = a;
    }
else
    {
    c = b;
    }

return;
}
```

Funzione minimo con parametro di tipo reference

Linguaggio C++: peculiarità – la classe **string**

il C++ semplifica grandemente l'uso delle stringhe per mezzo di una classe predefinita di library, la **classe string** appunto

```
#include <iostream>

using namespace std;

#include <string>

int main()
{
string parola;

cout<<"Fornisci una parola: ";
cin>>parola;
cout<<"La parola inserita è "<<parola<<"\n";

return 0;
}
```

Si osservi anzitutto la dichiarazione

#include <string>

posta all'inizio del programma e necessaria per poter utilizzare la classe **string**

A questo punto si dichiara un oggetto appartenente alla classe con la sintassi

string *nome_oggetto* ;

N.B. Non è necessario dichiarare una dimensione massima per l'oggetto **string** in quanto esse in C++ vengono gestite dinamicamente, riservando in modo automatico la memoria necessaria.

L'istruzione **cin**, usata nell'esempio precedente per acquisire una stringa, presenta il problema che non acquisisce stringhe contenenti al proprio interno degli spazi (**blank**)

Il problema può essere risolto in C++ usando la funzione **getline** nel seguente modo:

getline(cin, parola);

La sintassi è abbastanza semplice: **getline (cin, *nome_string_da_acquisire*)**

Linguaggio C++: peculiarità – la classe **string**

Il C++ permette svariate forme di inizializzazione di una variabile **string**

- **con l'operatore =** chiediamo al compilatore una inizializzazione con copia dell'oggetto
- **senza l'operatore =** chiediamo al compilatore la inizializzazione diretta

```
string s1;              // Inizializzazione default: stringa vuota
string s2(s1);          // Inizializzazione diretta: s2 copia di s1
string s2 = s1;         // Inizializzazione di copia, equivale a s2(s1)
string s1("Prova");     // Inizializzazione diretta
string s1 = "Prova";    // Inizializzazione di copia
```

Linguaggio C++: peculiarità – la classe **string**

Operatori

La classe string ridefinisce tutta una serie di operatori standard del C (overloading) in modo che siano applicabili a oggetti di tipo string. Gli operatori principali della classe sono:

- **Assegnazione** (=)
  E' possibile assegnare un valore a una stringa mediante l'uguale. Qui sotto vengono mostrati alcuni esempi:

```
string nome1, nome2;

nome1 = "pippo";
nome1 = nome2;
```

- **Concatenazione** (+)
  Concatenare due stringhe vuol dire creare una terza stringa formata dall'unione delle prime due:

```
string string1 = "uno due ";
string string2 = " tre quattro";

string string3 = string1 + string2;

// Visualizza "uno due tre quattro"
cout<<string3<<"\n";
```

  Si noti l'inizializzazione del valore delle stringhe contestualmente alla loro dichiarazione.

Linguaggio C++: peculiarità – la classe **string**

- **Operatori di confronto** (==, !=, <, <=, >, >=)
  Gli esempi seguenti dovrebbero essere sufficienti per illustrare l'uso degli operatori di confronto con le stringhe:

```
string parola1, parola2;

cout<<"Fornisci la prima parola: ";
cin>>parola1;
cout<<"Fornisci la seconda parola: ";
cin>>parola2;

if (parola1==parola2)
cout<<parola1<<" è uguale a "<<parola2<<"\n";

if (parola1!=parola2)
cout<<parola1<<" è diversa da "<<parola2<<"\n";

if (parola1<parola2)
cout<<parola1<<" precede "<<parola2<<" nell'ordine alfabetico\n";

if (parola1>parola2)
cout<<parola1<<" segue "<<parola2<<" nell'ordine alfabetico\n";
```

Linguaggio C++: peculiarità – la classe **string**

Principali metodi

I principali metodi della class string sono i seguenti:

- **Size e length**
  Il metodo *size* consente di determinare la lunghezza (numero di caratteri) di una stringa nel seguente modo (il numero di caratteri nell'esempio è 6):

```
string parola = "Torino";
cout<<"La stringa "<<parola<< " contiene "<<parola.size()<<" caratteri\n";
```

  Il metodo length funziona allo stesso modo e viene spesso usato in alternativa a size.

- **Substr**
  Il metodo *substr* restituisce una stringa ottenuta estraendo dalla stringa data il numero di caratteri specificato a partire dalla posizione indicata (0 indica il primo carattere della stringa). Il primo argomento tra parentesi indica la posizione da cui deve iniziare l'estrazione dei caratteri; il secondo argomento è il numero di caratteri da estrarre. L'esempio seguente dovrebbe chiarire l'applicazione del metodo:

```
string parola = "Torino";

// visualizza "Tori"
cout<<parola.substr(0,4)<<"\n";

// visualizza "rino"
cout<<parola.substr(2,4)<<"\n";
```

  Un operatore simile a size, che restituisce sempre il numero di caratteri della stringa è *length* (uso: *parola.length()*).

Linguaggio C++: peculiarità – la classe **string**

- **Replace**
  Il metodo *replace* sostituisce una sottostringa all'interno di una stringa con un'altra sottostringa (che può anche avere un numero diverso di caratteri). Il primo argomento tra parentesi indica la posizione da cui deve iniziare l'estrazione dei caratteri; il secondo argomento è il numero di caratteri da sostituire nella stringa di partenza; il terzo argomento è la sottostringa da sostituire. Esempio:

```
string parola = "Torino";

// visualizza "Milano"
cout<<parola.replace(0,4,"Mila")<<"\n";

// visualizza "Merano"
cout<<parola.replace(1,2,"er")<<"\n";

// visualizza "Loano"
cout<<parola.replace(0,3,"Lo")<<"\n";

// visualizza "Loreto"
cout<<parola.replace(2,3,"reto")<<"\n";
```

Linguaggio C++: peculiarità – la classe **string**

- **Accesso ai singoli caratteri**
  E' possibile accedere ai singoli caratteri che compongono una stringa come se fossero gli elementi di un vettore (in modo analogo a quanto avviene in C). Esempio:

```
string str ("stringa di prova");

for (int i=0; i<str.size(); ++i)
  cout << str[i];
```

  Il metodo length fornisce la lunghezza della stringa (numero di caratteri). Si presti attenzione al fatto che, sebbene sia possibile in C++ accedere ai caratteri di una stringa come se fossero gli elementi di un vettore di char, ciò non significa che un oggetto di tipo string *sia* un vettore di char (vedi qui sotto la spiegazione dell'uso di c_str).

  Una possibilità alternativa consiste nell'usare il metodo at, nel seguente modo:

```
string str ("stringa di prova");

for (int i=0; i<str.size(); ++i)
  cout << str.at(i);
```

  In entrambi i casi il risultato è lo stesso.

- **Conversione della stringa nello stile C (cioè come vettore di char)**
  In alcuni casi occorre convertire un oggetto di tipo string in un vettore di caratteri, cioè nella modalità con cui le stringhe vengono definite nel C tradizionale. Questo è utile in particolare quando è necessario passare una stringa a una funzione che richiede un vettore di char. In questi casi occorre usare la member function *c_str()* nel seguente modo:

```
string stringa ("stringa di prova");
cout<<strlen(stringa.c_str());
```

Linguaggio C++: OOP programming

In generale, in un **programma tradizionale** esiste una **funzione principale** (**main**) ed una serie di **funzioni secondarie** richiamate dalla stessa funzione principale

Tale tipo di approccio si chiama **top-down**, in quanto l'esecuzione va dall'alto verso il basso (ovvero parte dall'inizio della funzione principale e termina alla fine della stessa funzione)

Nella programmazione procedurale il codice e i dati restano sempre distinti

Le funzioni definiscono quello che deve accadere ai dati ma tali due elementi, codice e dati, non diventano mai una cosa sola

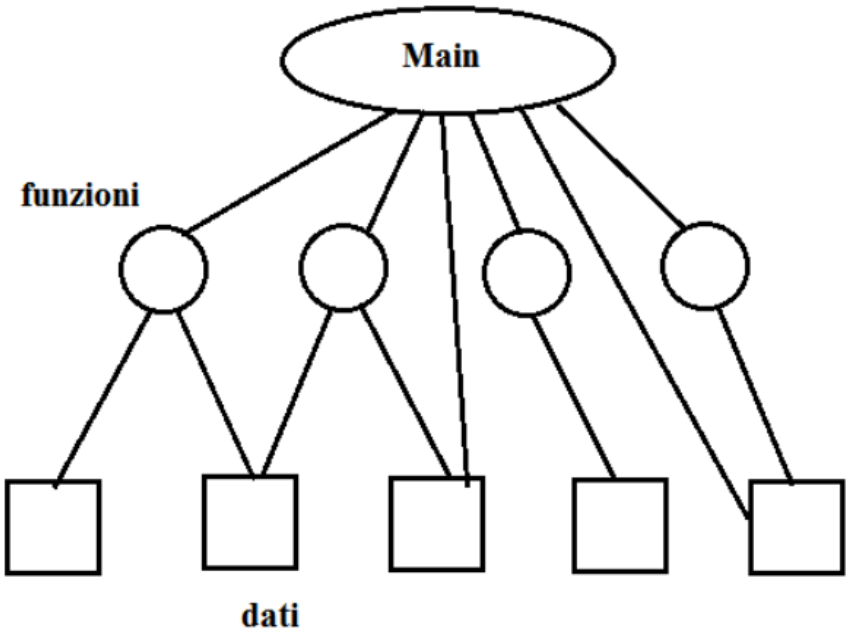
Uno degli **svantaggi principali** della programmazione procedurale è rappresentato dalla **manutenzione del programma**: spesso, per aggiungere o modificare parti di un programma **è necessaria la rielaborazione di tutto il programma stesso**

Questo approccio richiede un enorme quantità di tempo e di risorse che non è certamente un fattore trascurabile

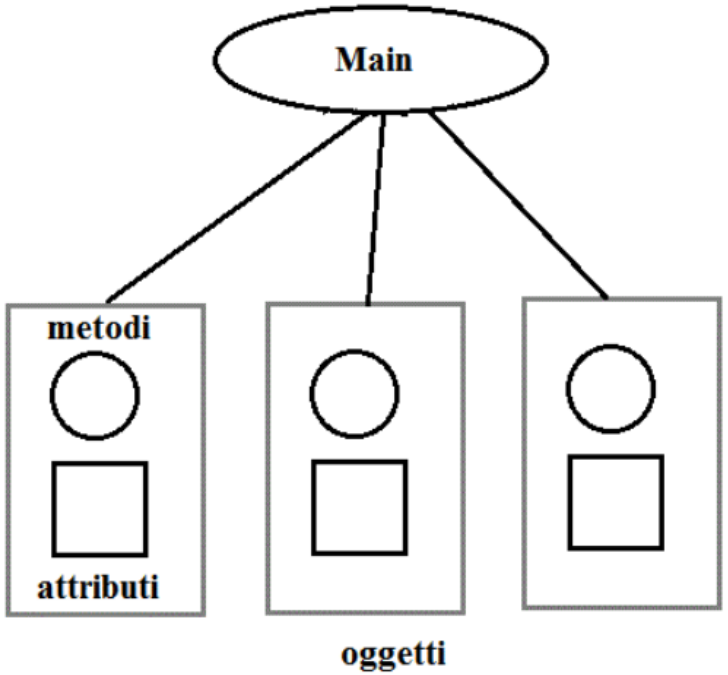
Linguaggio C++: OOP programming

- La programmazione ad oggetti rappresenta un ulteriore sviluppo rispetto alla programmazione modulare.
- La programmazione orientata agli oggetti (Object Oriented Programming, OOP) è un paradigma di programmazione, in cui un programma viene visto come un insieme di oggetti che interagiscono tra di loro.
- Nei linguaggi OOP esiste un nuovo tipo di dato, la classe. Questo tipo di dato serve appunto a modellare un insieme di oggetti dello stesso tipo.
- In generale, un oggetto è caratterizzato da un insieme di attributi e da un insieme di funzionalità

Linguaggio C++: OOP programming



Programmazione procedurale



Programmazione ad oggetti

Linguaggio C++: OOP programming

Un **programma orientato agli oggetti** funziona in maniera molto diversa rispetto ad un programma tradizionale

Vi sono, fondamentalmente, **tre vantaggi** per un programmatore:

Il primo vantaggio è la facilità di manutenzione del programma

I programmi risultano più semplici da leggere e da comprendere

Il secondo vantaggio è costituito dalla possibilità di modificare più facilmente il programma aggiungendo o modificando nuove funzionalità o cancellando operazioni non più necessarie

Per eseguire tali operazioni basta aggiungere o cancellare i relativi oggetti

I nuovi oggetti erediteranno le proprietà degli oggetti da cui derivano; sarà solo necessario aggiungere modificare o cancellare gli elementi differenti.

Il terzo vantaggio è dovuto al fatto che gli oggetti possono essere utilizzati più volte (riuso del software)

In definitiva, la programmazione ad oggetti rappresenta un modo di interpretare i concetti proprio come una serie di oggetti. Utilizzando, dunque gli oggetti, è possibile rappresentare le operazioni che devono essere eseguite e le loro eventuali interazioni

Linguaggio C++: OOP programming principi base

Il concetto che sta alla base della programmazione ad oggetti è quello della **classe**

Una classe C++ rappresenta un tipo di dati astratto che può contenere elementi in stretta relazione tra loro e che condividono gli stessi attributi

Un oggetto, di conseguenza, è semplicemente un'istanza di una classe

Le caratteristiche della classe vengono denominate **proprietà** o **attributi** , mentre le azioni sono dette **metodi o funzioni membro (member funcion)**

Esempio: consideriamo la **classe Animale**

Essa può essere vista come **un contenitore generico di dati e funzioni** i quali identificano le caratteristiche (es: il nome, la specie, ecc.) e le azioni (es: mangiare, dormire, ecc.) comuni a tutti gli animali

Una **istanza** della classe animale è rappresentata, ad esempio, **dall'oggetto** cane

Il cane è un animale con delle caratteristiche e delle azioni particolari che specificano in modo univoco le **proprietà** ed i **metodi** definiti nella **classe Animale**

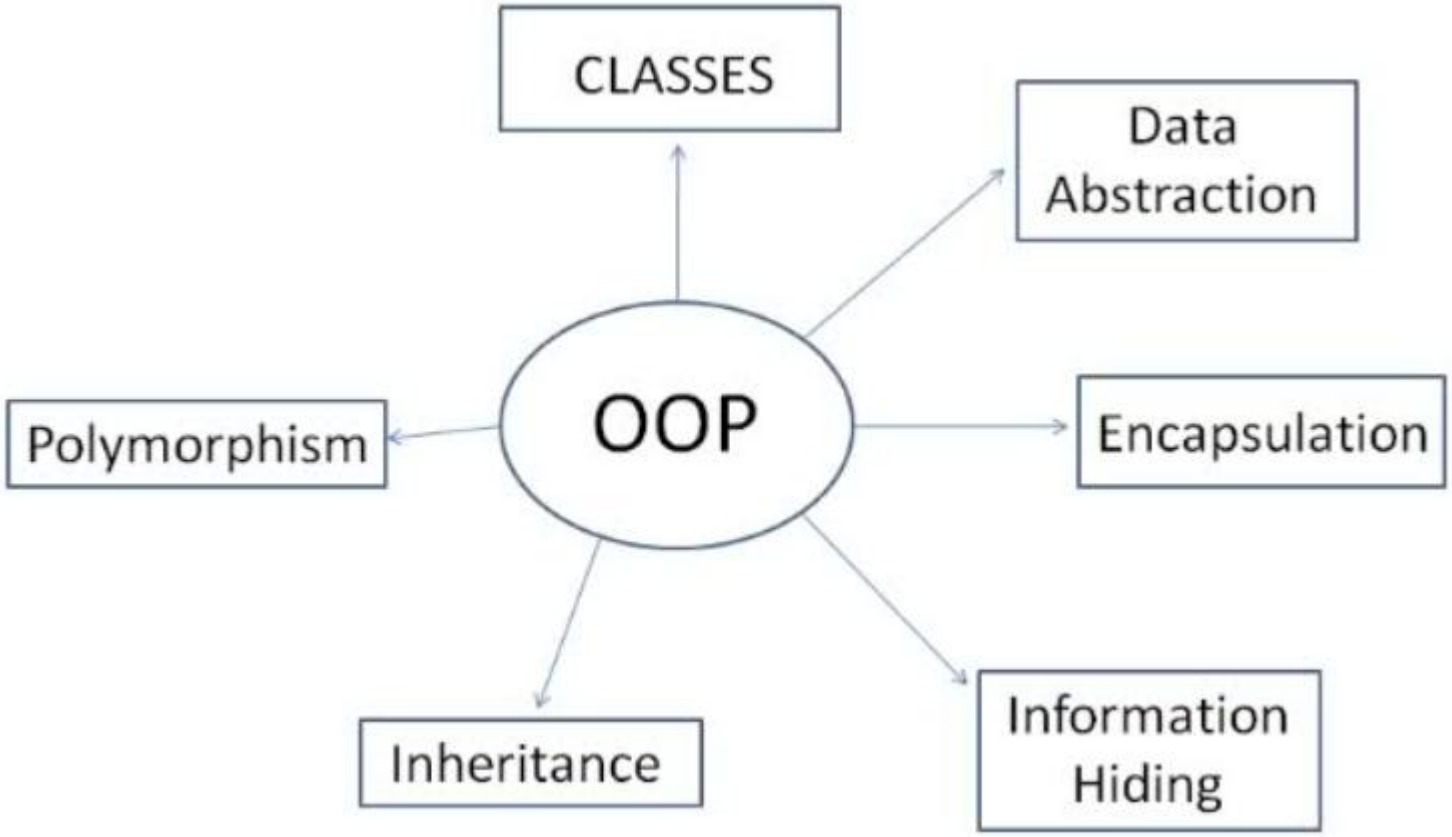
Linguaggio C++: OOP programming principi base

- Possibilità di dichiarare una funzione come funzione membro di una determinata classe, che significa che tale funzione appartiene strettamente a quella classe e può agire solo sui membri di quella classe o su quelle da essa derivate.
- Possibilità di dichiarare i singoli attributi e funzioni membro della classe come:
  private: accessibili solo alle funzioni membro appartenenti alla stessa classe
  protected: accessibili alle funzioni membro appartenenti alla stessa classe e alle classi da questa derivate (vedremo tra poco cosa significa classe derivata).
  public: accessibili da ogni parte del programma entro il campo di validità della classe in oggetto.
- Possibilità di definire in una stessa classe, dati (attributi e/o funzioni membro) con lo stesso identificatore ma con diverso campo di accessibilità (public, protected, public). Tuttavia, si sconsiglia di usare questo approccio per non rendere difficilmente leggibile il programma.

Linguaggio C++: OOP programming principi base

- Possibilità di overloading delle funzioni (si veda la definizione data nei capitoli precedenti).
- **Incapsulamento**. Con tale termine si definisce la possibilità offerta dal C++ di collegare strettamente i dati contenuti in una classe con le funzioni che la manipolano. L'oggetto, è proprio l'entità logica che deriva dall'incapsulamento.
- **Ereditarietà**. E' la possibilità per un oggetto di acquisire le caratteristiche (attributi e funzioni membro) di un altro oggetto.
- **Polimorfismo**. Rappresenta la possibilità di utilizzare uno stesso identificatore per definire dati (attributi) o operazioni (funzioni membro) diverse allo stesso modo di come, per esempio, animali appartenenti ad una stessa classe possono assumere forme diverse in relazioni all'ambiente in cui vivono.

Linguaggio C++: OOP programming principi base



Linguaggio C++: definizione di una classe

Per **definire una classe in C++** utilizziamo la parola riservata **class**

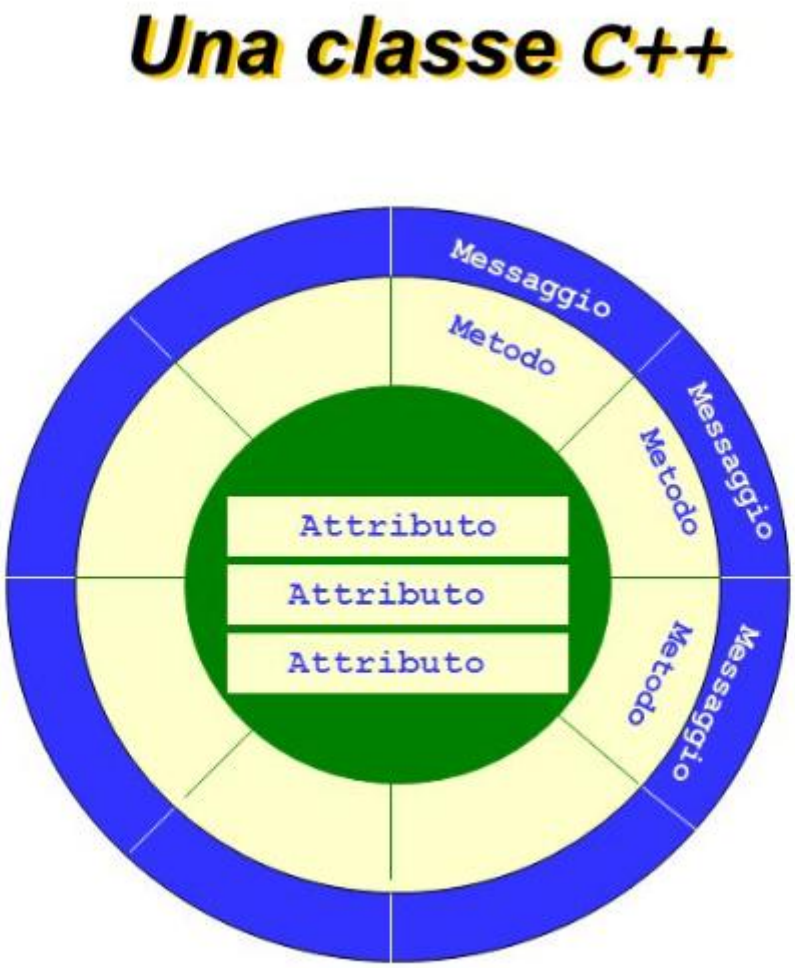
Essa ci permette di definire l'*interfaccia* della classe, ovvero le proprietà e i metodi che gli oggetti metteranno a disposizione (esporranno) all'esterno

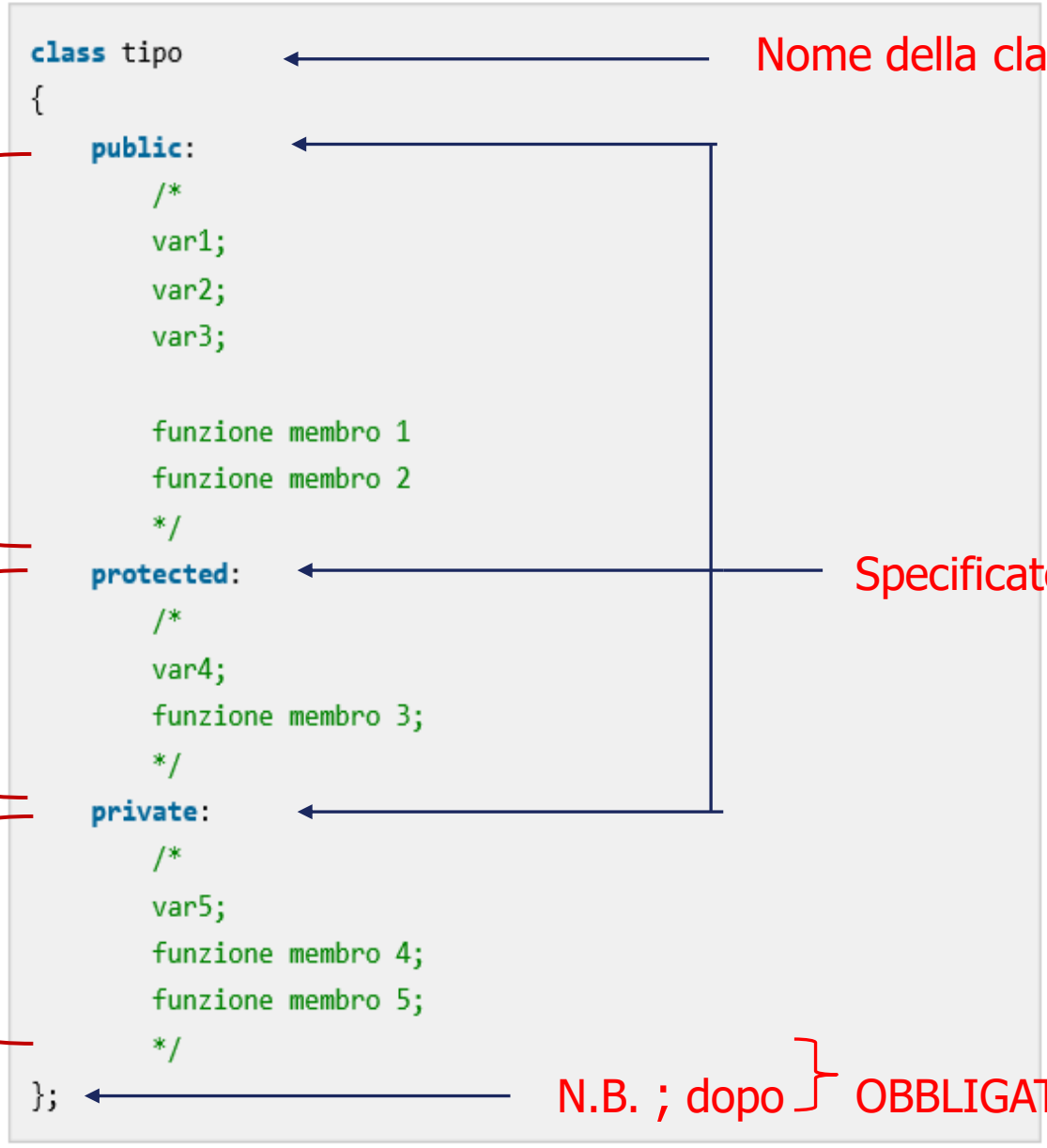
Oltre a ciò nel blocco che segue **class** dichiariamo anche gli elementi **protetti** e **privati** dei nostri oggetti

Subito dopo **class** indichiamo il **nome della classe** (ovvero il **nome del tipo**) e procediamo alla **definizione** della sua **struttura**

Vediamo ora un esempio in cui definiamo una classe e indichiamo con i commenti una possibile organizzazione dei suoi elementi

Linguaggio C++: disegno di una classe





la sezione **public** contiene membri (attributi e/o metodi) a cui si può accedere dall'esterno della classe

la sezione **protected** contiene membri a cui si può accedere anche da metodi di classi *derivate*

la sezione **private** contiene membri a cui si può accedere solo dall'interno della classe

N.B. Se non specifichiamo nessun modificatore per un metodo o per un attributo all'interno di una classe, questi si intendono automaticamente private

Linguaggio C++: definizione di una classe

Progettiamo ora il primo esempio di classe: **Cliente**

Utilizzando il diagramma delle classi previsto da linguaggio UML possiamo scrivere:

Cliente
+ Nome: **string** + Cognome: **string** + Indirizzo: **string**
+ **PROCEDURA** set_Nome (**VAL** s: **string**) + **FUNZIONE** get_Nome ( ) : **string** + **PROCEDURA** set_Cognome (**VAL** s: **string**) + **FUNZIONE** get_Cognome ( ) : **string** + **PROCEDURA** set_Indirizzo (**VAL** s: **string**) + **FUNZIONE** get_Indirizzo ( ) : **string**

Linguaggio C++: definizione di una classe

Implementiamo in C++ ora la classe **Cliente**

```
class Cliente
{
public:
    string Nome;
    string Cognome;
    string Indirizzo;
    void set_Nome(string);
    string get_Nome( );
    void set_Cognome(string);
    string get_Cognome( );
    void set_Indirizzo(string);
    string get_Indirizzo( );
};
```

N.B. Ovviamente costruire una classe con soli membri pubblici **non realizza appieno** uno dei principi basilari dell'OOP ossia **l'information hiding** ed è **dunque una pratica da sconsigliare**

In questa classe abbiamo definito sia gli attributi, sia le funzioni membro come public, ovvero accessibili dall'esterno e in ogni punto del programma

Salviamo questa definizione di *interfaccia* in un file chiamato **Cliente.h** e creiamo un nuovo file, che chiamiamo **Cliente-main.cpp**, in cui scriviamo l'*implementazione* dei metodi

Linguaggio C++: definizione di una classe

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

void Cliente::set_Nome(string s)
{
Nome = s;
return;
}

string Cliente::get_Nome( )
{
return Nome;
}

void Cliente::set_Cognome(string s)
{
Cognome = s;
return;
}

string Cliente::get_Cognome( )
{
return Cognome;
}

void Cliente::set_Indirizzo(string s)
{
Indirizzo = s;
return;
}

string Cliente::get_Indirizzo( )
{
return Indirizzo;
}
```

Il file **Cliente-main.cpp** potrebbe essere fatto così:

Avrete certamente notato la particolare sintassi utilizzata nel file **Cliente-main.cpp** relativamente alla implementazione delle funzioni membro. Essa segue la regola:

tipo_restituito nome_classe::nome_metodo (eventuali parametri)

Linguaggio C++: definizione di una classe

```
int main(int argc, char** argv)
{
Cliente c;
string s1, s2, s3;

cout << "Nome : ";
getline(cin,s1);
c.set_Nome(s1);
cout << "Cognome : ";
getline(cin,s2);
c.set_Cognome(s2);
cout << "Indirizzo : ";
getline(cin,s3);
c.set_Indirizzo(s3);

cout << "Il nome del cliente inserito e': " << c.get_Nome() << endl;
cout << "Il cognome del cliente inserito e': " << c.get_Cognome() << endl;
cout << "L' indirizzo del cliente inserito e': " << c.get_Indirizzo() << endl;
return 0;
}
```

Definizione di un oggetto della classe Cliente (STACK)

Accesso ai suoi metodi tramite il nome dell'oggetto.
Operatore **punto .**

Avrete certamente notato la particolare sintassi utilizzata nel file **Cliente-main.cpp** relativamente alla implementazione delle funzioni membro. Essa segue la regola:

```
tipo_restituito nome_classe::nome_metodo( eventuali parametri)
```

Linguaggio C++: definizione di una classe

Supponiamo di utilizzare un oggetto della classe Cliente allocato dinamicamente

```
int main(int argc, char** argv)
{
Cliente* c1 = new Cliente();
string s1, s2, s3;

cout << "Nome : ";
getline(cin,s1);
c1->set_Nome(s1);
cout << "Cognome : ";
getline(cin,s2);
c1->set_Cognome(s2);
cout << "Indirizzo : ";
getline(cin,s3);
c1->set_Indirizzo(s3);

cout << "Il nome del cliente inserito e': " << c1->get_Nome() << endl;
cout << "Il cognome del cliente inserito e': " << c1->get_Cognome() << endl;
cout << "L' indirizzo del cliente inserito e': " << c1->get_Indirizzo() << endl;

delete(c1);

return 0;
}
```

Definizione di un puntatore ad un oggetto della classe Cliente

Allocazione dinamica (HEAP) di un oggetto della classe Cliente

Accesso ai suoi metodi tramite puntatore ad un oggetto
Operatore freccia **->**

Deallocazione dinamica (HEAP) di un oggetto della classe Cliente

Linguaggio C++: definizione di una classe

In questo caso nella funzione main definiamo un oggetto della classe `Cliente` servendoci però di un puntatore. In questo caso entrano in gioco due keyword fondamentali

- **new**, che alloca la memoria necessaria all'instanziazione dell'oggetto e ne ritorna la relativa locazione di memoria.
- **delete**, che servirà per liberare la memoria utilizzata per l'oggetto, una volta che non ci servirà più

Avendo a che fare con un puntatore all'oggetto, cambierà anche la modalità con la quale facciamo riferimento ai suoi metodi o ai suoi attributi (o proprietà). In questo caso infatti, invece del punto utilizziuamo l'operatore freccia (`->`):

```
oggetto->attributo;
oggetto->metodo();
```

Linguaggio C++: definizione di una classe

Trasformiamo ora il primo esempio di classe: **Cliente**

Utilizzando il diagramma delle classi previsto da linguaggio UML possiamo scrivere:

Cliente
- Nome: **string** - Cognome: **string** - Indirizzo: **string**
+ **PROCEDURA** set_Nome (**VAL** s: **string**) + **FUNZIONE** get_Nome ( ) : **string** + **PROCEDURA** set_Cognome (**VAL** s: **string**) + **FUNZIONE** get_Cognome ( ) : **string** + **PROCEDURA** set_Indirizzo (**VAL** s: **string**) + **FUNZIONE** get_Indirizzo ( ) : **string**

Linguaggio C++: definizione di una classe

Se non specifichiamo nessun modificatore per un metodo o per un attributo all'interno di una classe, questi si intendono automaticamente `private`. Ad esempio:

```
class Cliente
{
    string Nome;
    string Cognome;
    string Indirizzo;
public:
    void set_Nome(string);
    string get_Nome( );

    void set_Cognome(string);
    string get_Cognome( );

    void set_Indirizzo(string);
    string get_Indirizzo( );
};
```

Sono privati anche se non specificato private

gli identificatori: `nome`, `cognome` e `indirizzo`, che si trovano subito dopo la dichiarazione del nome della classe, saranno attributi `private` e, come tali, utilizzabili soltanto all'interno della classe stessa.

Linguaggio C++: definizione di una classe

Dobbiamo quindi fare attenzione e ricordare di inserire i modificatori di visibilità per metodi e attributi di una classe. Per non trovarci in una situazione simile a questa:

```
class Cliente
{
    string Nome;
    string Cognome;
    string Indirizzo;

    void set_Nome(string);
    string get_Nome( );

    void set_Cognome(string);
    string get_Cognome( );

    void set_Indirizzo(string);
    string get_Indirizzo( );
};
```

In questo caso infatti la classe `Cliente` sarebbe assolutamente inutilizzabile dall'esterno visto che tutti i suoi dati e metodi sono privati (non abbiamo specificato alcun modificatore di accesso). Per fortuna questo sarebbe chiaramente un errore e anche il compilatore ce lo farebbe notare.

Linguaggio C++: **costruttore** di una classe

Costruttore

Possiamo considerare il **costruttore** come una particolare funzione membro di una classe, che è eseguita ogni volta che viene creato un nuovo oggetto della classe cui appartiene.

In questo modo, tramite il costruttore possiamo *determinare il comportamento che l'oggetto avrà alla sua creazione*: ad esempio possiamo utilizzare i costruttori per inizializzare le variabili della classe o per allocare aree di memoria.

Qualche nota tecnica iniziale, prima di passare alla sintassi e algi esempi:

- il costruttore di una classe deve sempre avere lo stesso nome della classe in cui è definito;
- il costruttore può accettare argomenti e possono essere modificato tramite overloading;
- se non dichiarato esplicitamente all'interno della classe, il costruttore viene generato automaticamente dal compilatore (costruttore di default).

Linguaggio C++: **distruttore** di una classe

Distruttore

Come è facile immaginare il distruttore ha una funzione simile ma opposta al costruttore: anch'esso è una particolare funzione membro che però viene eseguita automaticamente quando stiamo per rilasciare un oggetto, tramite l'operatore `delete` ad un puntatore all'oggetto, il programma esce dal campo di visibilità di un oggetto della classe.

Anche in questo caso ci permette di gestire comportamenti come il rilascio al sistema della memoria allocata internamente dall'oggetto.

- Il distruttore, come il costruttore, ha sempre lo stesso nome della classe nella quale è definito ma è preceduto dal carattere tilde (~);
- a differenza dei costruttori, i distruttori non possono accettare argomenti e non possono essere modificati tramite overloading.
- anche i distruttori, quando non vengono definiti esplicitamente, vengono creati automaticamente dal compilatore (distruttore di default).

Linguaggio C++: **costruttore** e **distruttore** di una classe

Esercizio: Proviamo ad implementare la versione definitiva della classe **Cliente** provando a dichiarare ed a valorizzare più oggetti sia staticamente sia dinamicamente utilizzando entrambi i costruttori

Cliente
- Nome: **string**
- Cognome: **string**
- Indirizzo: **string**
+ **PROCEDURA** set_Nome (**VAL** s: **string**)
+ **FUNZIONE** get_Nome ( ) : **string**
+ **PROCEDURA** set_Cognome (**VAL** s: **string**)
+ **FUNZIONE** get_Cognome ( ) : **string**
+ **PROCEDURA** set_Indirizzo (**VAL** s: **string**)
+ **FUNZIONE** get_Indirizzo ( ) : **string**
+ **COSTRUTTORE** **Cliente** ( )
+ **COSTRUTTORE** **Cliente** (**VAL** s1: **string** , **VAL** s2: **string** , **VAL** s3: **string**)
+ **DISTRUTTORE** **~Cliente** ( )

Linguaggio C++: esempio conversione 1

Vediamo un semplice programma che fa uso del costruttore e del distruttore. Un semplice programma che converte le lire in Euro.

```
#include <iostream>

using namespace std;

class Conversione
{
    public:
        long valore_lira;
        float valore_euro;

        void ottieni_valore();
        float converti_lira_in_euro( );

        Conversione();     // un solo costruttore
        ~Conversione();    // il distruttore
};
```

Linguaggio C++: esempio conversione 1

```
void Conversione::ottieni_valore()
{
    cout << "Inserire il valore in lire: " ;
    cin >> valore_lira;
    cout << endl;
}

float Conversione::converti_lira_in_euro()
{
    float risultato;
    risultato = ((float) valore_lira) / (float) 1936.27;
    return risultato;
}

// Costruttore
Conversione::Conversione()
{
    cout << "Costruttore: Inizio della conversione\n";

    valore_lira = 0;    // Inizializza valore_lira
    valore_euro = 0.0;  // Inizializza valore_euro
}

// Distruttore
Conversione::~Conversione( )
{
    cout << "Distruttore: Fine della conversione\n";
}
```

Linguaggio C++: esempio conversione 1

```
int main(int argc, char** argv)
{
Conversione conv;

conv.ottieni_valore( );

conv.valore_euro = conv.converti_lira_in_euro( );

cout << conv.valore_lira << " in lire, vale " << conv.valore_euro << " Euro.";

cout << endl;

return(0);
}
```

Linguaggio C++: esempio conversione 1

Se si prova ad eseguire il programma precedente, si noterà che la prima cosa che viene stampata sullo schermo è:

```
Inizio della conversione.
```

Che corrisponde all'istruzione eseguita all'interno del costruttore della classe `Conversione`. Il costruttore della classe viene invocato non appena si costruisce un'istanza della classe stessa, ovvero quando nel `main` viene eseguita l'istruzione:

```
Conversione conv;
```

Nel il costruttore sono contenute due istruzioni di inizializzazione di variabili della classe. È buona norma inserire tutte le inizializzazioni sempre all'interno del costruttore della classe.

Non è buona norma invece inserire l'output a schermo all'interno del costruttore, lo abbiamo fatto qui per esplicitare il momento in cui il costruttore viene richiamato.

Linguaggio C++: esempio conversione 1

L'ultima stampa sullo schermo sarà:

```
Fine della conversione.
```

Che corrisponde all'invocazione del distruttore della classe `Conversione`. La chiamata del distruttore viene effettuata in modo automatico non appena l'oggetto costruito precedentemente esce dallo scopo del programma (in tal caso proprio alla fine del programma stesso).

Linguaggio C++: esempio conversione 2

Allocare e rilasciare memoria degli oggetti

Vediamo ora come allocare e rilasciare spazi di memoria rispettivamente nel costruttore e nel distruttore. Modifichiamo leggermente l'esempio precedente:

```
#include <iostream>

using namespace std;


class Conversione
{
    public:
        long* valore_lira;
        float valore_euro;

        void ottieni_valore();
        float converti_lira_in_euro( );

        Conversione();     // un solo costruttore
        ~Conversione();    // il distruttore
};
```

La variabile **valore_lira** è stata definita ora come puntatore. Perciò è necessario allocare spazio per utilizzarla e questa allocazione, come buona norma insegna, viene effettuata nel **costruttore** della classe

Linguaggio C++: esempio conversione 2

La variabile **valore_lira** è stata definita come puntatore. Perciò è necessario allocare spazio per utilizzarla e questa allocazione, come buona norma insegna, viene effettuata nel **costruttore** della classe

```
Conversione::Conversione()
{
    cout << "Costruttore: Inizio della conversione\n";

    valore_lira = new long(0);     // Inizializza valore_lira
    valore_euro = 0.0;             // Inizializza valore_euro
}

// Distruttore
Conversione::~Conversione( )
{
    cout << "Distruttore: Fine della conversione\n";
    delete (valore_lira);
}
```

Una volta allocata la memoria occorre ricordarsi di rilasciarla al sistema quando non occorre più al programma. Il **distruttore** rappresenta il punto ideale per rilasciare la memoria dinamica utilizzata per le variabili come **valore_lira**.

Linguaggio C++: esempio conversione 2

```
void Conversione::ottieni_valore()
{
    cout << "Inserire il valore in lire: " ;
    cin >> *valore_lira;
    cout << endl;
}

float Conversione::converti_lira_in_euro()
{
    float risultato;
    risultato = ((float) *valore_lira) / (float) 1936.27;
    return risultato;
}

int main(int argc, char** argv)
{
Conversione conv;

conv.ottieni_valore( );
conv.valore_euro = conv.converti_lira_in_euro( );
cout << *conv.valore_lira << " in lire, vale " << conv.valore_euro << " Euro.";
cout << endl;

return(0);
}
```

Utilizzo l'operatore * applicato alla variabile di tipo puntatore **valore_lira** per fornire in valore in lire da convertire

Linguaggio C++: il puntatore this

La parola chiave **this** identifica un puntatore che fa riferimento alla classe. Non occorre dichiararlo poichè la sua dichiarazione è implicita nella classe.

```
nome_classe* this; // dove nome_classe è il tipo della classe
```

Il puntatore `this` punta all'oggetto per il quale è stata richiamata la funzione membro. Facciamo un esempio, data la seguente definizione di classe:

```
class Echo
{
private:
    char chr;

public:
    void set_char(char k);
    char get_char();
};
```

Linguaggio C++: il puntatore this

ecco l'utilizzo del puntatore `this` all'interno del metodo `get_char()`:

```
void Echo::set_char(char k)
{
chr = k;
}

char Echo::get_char()
{
return this->chr;
}
```

In questo caso, `this` consente di accedere alla variabile membro `chr`, una variabile privata della classe. Naturalmente `this` può avere tanti altri utilizzi, ma è importante sapere che esse rappresenta un puntatore alla classe che si sta utilizzando.

Linguaggio C++: ereditarietà

Come abbiamo detto, una **classe derivata** può essere considerata un'**estensione** di una classe oppure una classe che eredita le proprietà e i metodi da un'altra classe. La classe originaria viene denominata classe base mentre la classe derivata viene anche chiamata **sottoclasse** (o classe figlia).

Fondamentalmente, una classe derivata consente di espandere o personalizzare le funzionalità di una classe base, senza costringere a modificare la classe base stessa.

È possibile derivare più classi da una singola classe base e la **classe base** può essere una qualsiasi classe C++ e tutte le classi derivate ne rifletteranno la descrizione.

In genere, la classe derivata aggiunge nuove funzionalità alla classe base. Ad esempio, la classe derivata può modificare i privilegi d'accesso, aggiungere nuove funzioni membro o modificare tramite overloading le funzioni membro esistenti.

Linguaggio C++: ereditarietà

La sintassi di una classe derivata

Per descrivere una classe derivata si fa uso della seguente sintassi:

```
class classe-derivata : <specificatore d'accesso> classe-base
{
    ...
};
```

Ad esempio:

```
class pesce_rosso : public pesce
{
    ...
};
```

In tal caso, la classe derivata si chiama `pesce_rosso`. La classe base ha visibilità pubblica e si chiama `pesce`.

Linguaggio C++: ereditarietà

Il meccanismo di ereditarietà, pur essendo abbastanza semplice, richiede una certa attenzione per non cadere in errori perchè dipende dallo standard della classe base.

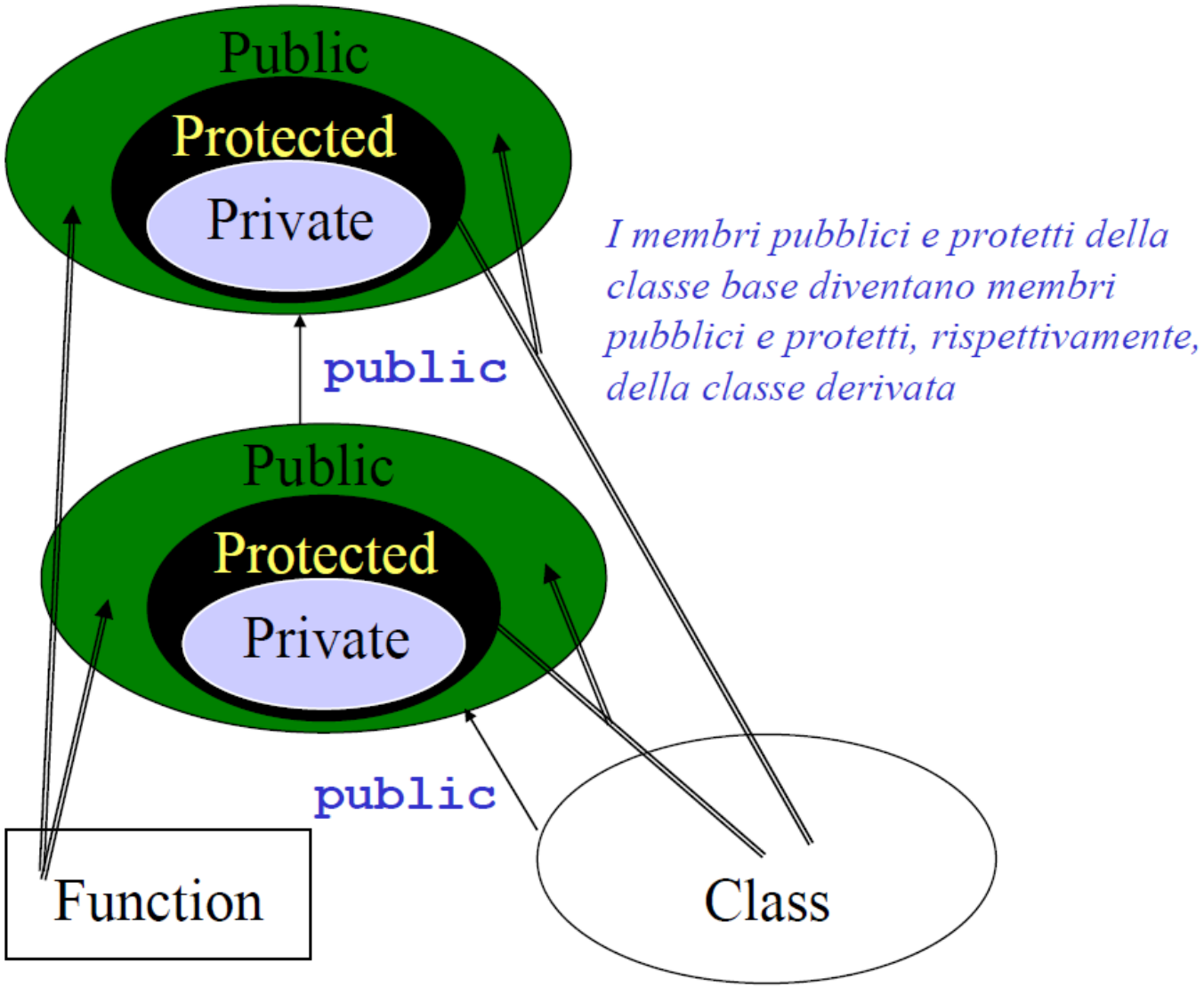
Gli attibuti ed i membri che vengono ereditati dalla classe base possono cambiare la loro visibilità nella classe figlia, in base allo specificatore d'accesso con il quale si esegue l'ereditarietà stessa.

- L'ereditarietà si presenta in tre distinte forme:

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

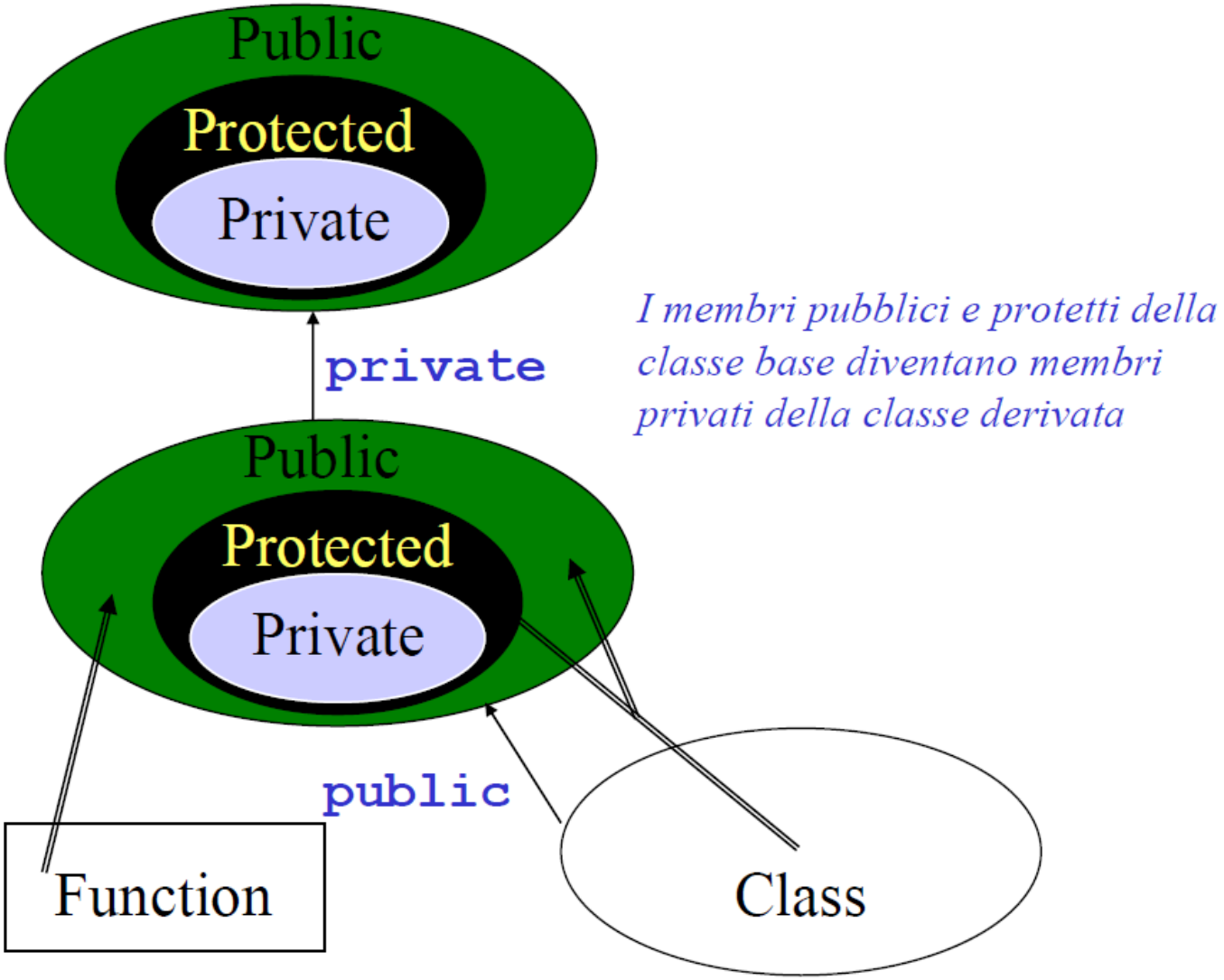
- Nell'*ereditarietà pubblica*, i membri ereditati hanno la stessa protezione che avevano nella classe base
  - gli utenti della classe derivata possono usare i membri pubblici ereditati
- Nell'*ereditarietà privata*, i membri ereditati divengono membri privati della classe ereditata
  - gli utenti della classe derivata non possono usare i membri ereditati
- Nell'*ereditarietà protetta*, i membri pubblici e protetti ereditati divengono membri protetti della classe derivata

Linguaggio C++: ereditarietà "public"



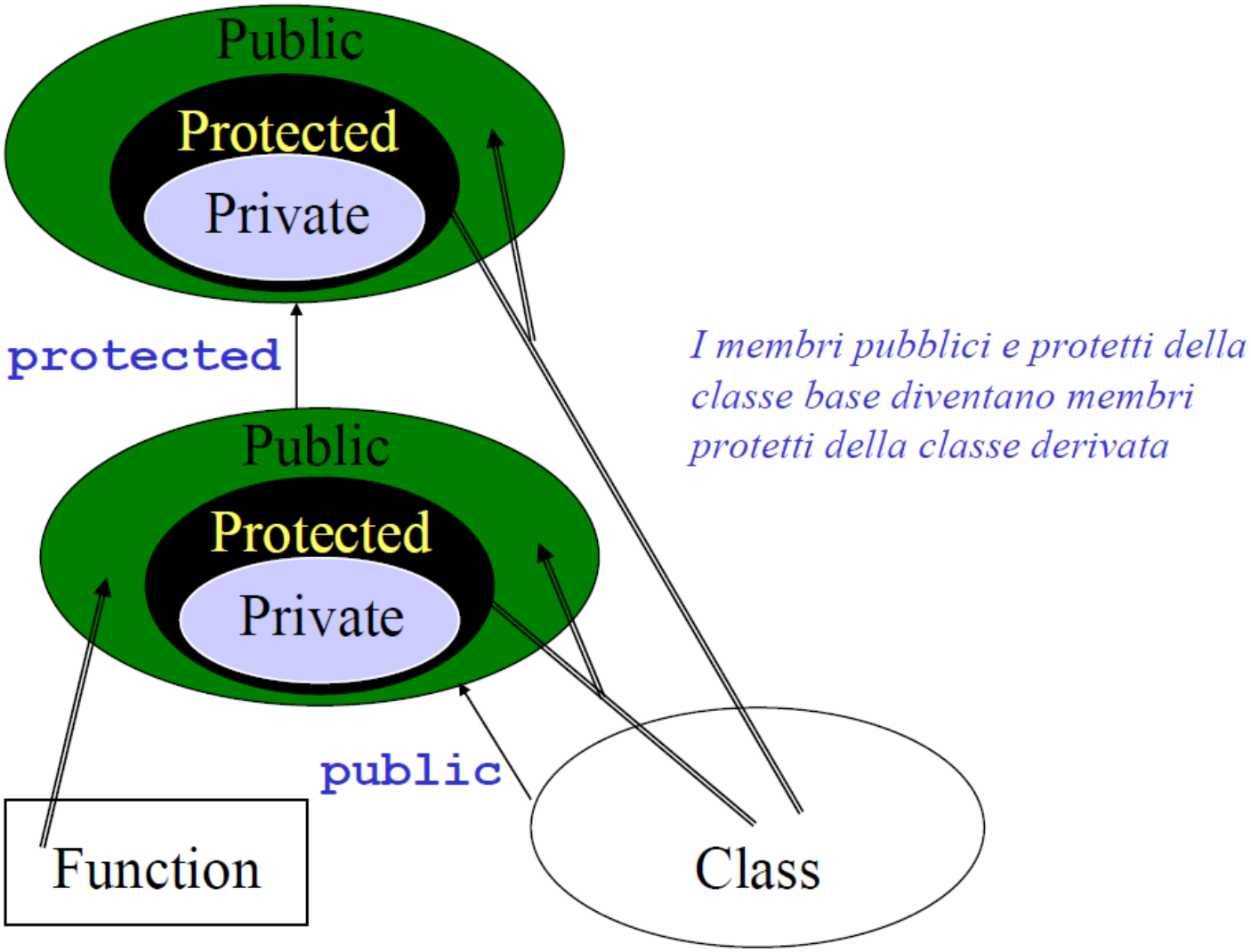
I membri pubblici e protetti della classe base diventano membri pubblici e protetti, rispettivamente, della classe derivata

Linguaggio C++: ereditarietà "private"



I membri pubblici e protetti della classe base diventano membri privati della classe derivata

Linguaggio C++: ereditarietà “protected”



Linguaggio C++: ereditarietà

Tipo di ereditarietà	*Classe base*	*Classe derivata*

`public`	`public`
	`protected`
	`private`
		`public`
		`protected`
		inaccessibile

`protected`	`public`
	`protected`
	`private`
		`protected`
		`protected`
		inaccessibile

`private`	`public`
	`protected`
	`private`
		`private`
		`private`
		inaccessibile

Linguaggio C++: esempio di ereditarietà

Vediamo ora un esempio che fa uso di una classe derivata: definiamo la classe `animale` e da essa facciamo discendere la classe `cane`.

Iniziamo con la definizione della classe base `animale`:

```
#include <iostream>
#include <string.h>

using namespace std;


class Animale
{
    protected:
        char specie[20];
        int eta;
        char sesso;

        void mangia ( );
        void beve ( );

    public:
        void ottieni_dati ( );

        Animale( );        // costruttore
        ~Animale( );       // distruttore
};
```

Include libreria **string.h** necessario per l'utilizzo delle funzioni **strcpy** e **toupper**

Membri ed attributi che saranno **accessibili** anche dalle classi derivate

Linguaggio C++: esempio di ereditarietà

```
void Animale::mangia()
{
    cout << "Invocato il metodo mangia\n";
}

void Animale::beve()
{
    cout << "Invocato il metodo beve\n";
}

void Animale::ottieni_dati()
{
    cout << "Inserire l'eta' dell'animale: ";
    cin >> eta;

    cout << "Inserire il sesso dell'animale (M o F): ";
    cin >> sesso;
}

Animale::Animale()
{
    strcpy(specie," ");
    cout << "Costruttore di un oggetto della classe Animale\n";
}

Animale::~Animale()
{
    cout << "Distruttore di un oggetto della classe Animale\n";
}
```

Linguaggio C++: esempio di ereditarietà

Definiamo ora la classe derivata `cane`:

```
class Cane : public Animale
{
    private:
        void abbaia();

    public:
        void esegui_azioni();
        void stampa_dati();

        Cane();             // costruttore
        ~Cane();            // distruttore
};
```

Membro privato che **"estende" (ESTENSIONE)** le funzionalità della classe derivata Cane

Linguaggio C++: esempio di ereditarietà

```
void Cane::abbaia()
{
    cout << "Invocato il metodo abbaia\n";
}

void Cane::stampa_dati()
{
    cout << "La specie dell'animale e': " << specie << endl;
    cout << "L' eta' dell'animale e': " << eta << endl;
    cout << "Il sesso dell'animale e' :" << (char)toupper(sesso) << endl;
}

void Cane::esegui_azioni()
{
    mangia();
    beve();
    abbaia();
}

Cane::Cane()
{
    cout << "Costruttore di un oggetto della classe Cane\n";
    strcpy(specie,"cane");
}

Cane::~Cane()
{
    cout << "Distruttore di un oggetto della classe Cane\n";
}
```

Metodo proprio della classe Cane

Metodi EREDITATI dalla **classe base Animale**

Metodo proprio **della classe derivata Cane**

Linguaggio C++: esempio di ereditarietà

```
int main(int argc, char** argv)
{
Cane c;

c.ottieni_dati();

c.stampa_dati();

c.esegui_azioni();

return (0);
}
```

Metodi EREDITATI dalla **classe base Animale**

Metodi propri della **classe derivata Cane**

Linguaggio C++: esempio di ereditarietà

È utile esaminare questo semplice programma per capire il funzionamento delle classi derivate. Come si può osservare, abbiamo creato una classe base, `animale`, che contiene alcune informazioni di base (metodi e attributi) comuni a tutti gli animali. Poi, abbiamo costruito una classe figlia, `cane`, che eredita tutte le informazioni della classe animale (che sono state dichiarate `protected`) ed in più implementa un nuovo metodo, `abbaia()`, comune soltanto alla specie `cane`.

Infine è utile notare anche l'utilizzo della funzione `strcpy`, implementata nella libreria standard `string.h`, che serve per copiare il valore di una stringa sorgente ad una destinazione.

Linguaggio C++: polimorfismo – l'OVERLOAD

L'overloading delle funzioni è una funzionalità specifica del C++ che non è presente in C e che può essere usata **anche senza utilizzare le classi**

Questa funzionalità permette di poter utilizzare lo **stesso nome** per una funzione e/o un metodo più volte all'interno dello stesso programma e/o classe, **a patto però che gli argomenti forniti (ossia la lista dei parametri) siano differenti** (segnature differenti nel numero e/o nel tipo dei parametri)

In maniera automatica, il programma eseguirà la funzione appropriata a seconda del tipo di argomenti passati

Ciò permette, di fatto, di avere all'interno di un programma o di una classe più metodi che hanno lo stesso nome, ma con liste di parametri diversi

N.B. Vedi esempio funzione **moltiplica** slide precedenti

Linguaggio C++: polimorfismo – l'OVERRIDE

Il polimorfismo in C++ si realizza oltre che con l'overloading delle funzioni anche tramite l'**overriding delle funzioni**

Con il termire **override** si intende una vera e propria riscrittura di un certo metodo di una classe che abbiamo ereditato

I metodi sia della classe base sia della classe derivata a cui è sato applicato **l'override** manterranno lo stesso nome e la stessa lista di parametri (stessa segnatura ossia stesso numero e stesso tipo) ma con un'implementazione differente

Con **l'override**, quindi, è possibile ridefinire un metodo di una classe base (classe generalista) adattandolo così alla classe derivata (classe specialista) mantenendo comunque una coerenza per quanto riguarda la semantica del metodo che avrà lo stesso identico nome e la stessa lista dei parametri

Linguaggio C++: polimorfismo – l'OVERRIDE (esempio)

Per capire il significato vero e proprio del meccanismo di **OVERRIDE** di un metodo utilizziamo un esempio basato su una classe "Dipendente" così strutturata
(N.B. per semplicità sono stati omessi i metodi ***set*** e ***get*** per gli attributi "nome" e "cognome")

```
class Dipendente
{
private:
          string nome;
          string cognome;
          int oreLavorativeMensili;
          int retribuzioneOraria;

protected:
         int stipendioBase;

public:
        void set_OreLavorativeMensili(int h);
        int get_OreLavorativeMensili();

        void set_RetribuzioneOraria(int e);
        int get_RetribuzioneOraria();

        int calcolaStipendio();

};
```

N.B. necessario per potervi accedere dalla classe derivata **ResponsabileDiProgetto**

N.B. metodo che subirà **l'OVERRIDE** nella classe derivata **ResponsabileDiProgetto**

Linguaggio C++: polimorfismo – l'OVERRIDE (esempio)

```
void Dipendente::set_OreLavorativeMensili(int h)
{
oreLavorativeMensili = h;
}

int Dipendente::get_OreLavorativeMensili()
{
return oreLavorativeMensili;
}

void Dipendente::set_RetribuzioneOraria (int e)
{
retribuzioneOraria = e;
}

int Dipendente::get_RetribuzioneOraria ()
{
return retribuzioneOraria;
}

int Dipendente::calcolaStipendio()
{
stipendioBase = oreLavorativeMensili * retribuzioneOraria;
return stipendioBase;
}
```

N.B. il metodo calcola lo stipendio base di una qualunque istanza (oggetto) appartenente alla classe **Dipendente**

Linguaggio C++: polimorfismo – l'OVERRIDE (esempio)

```
class ResponsabileDiProgetto: public Dipendente
{
private:
        int bonus;

public:
        void set_bonus(int b);
        int get_bonus();

        int calcolaStipendio();
};

void ResponsabileDiProgetto::set_bonus(int b)
{
bonus = b;
}

int ResponsabileDiProgetto::get_bonus()
{
return bonus;
}

int ResponsabileDiProgetto::calcolaStipendio()
{
int s;
s = get_OreLavorativeMensili() * get_RetribuzioneOraria() + bonus;
return s;
}
```

N.B. il metodo **calcolaStipendio ( )** della classe derivata **ResponsabileDiProgetto ha la stessa segnatura** dell'omonimo metodo della classe **Dipendente (OVERRIDE)**

N.B. il metodo **calcolaStipendio ( )** si specializza tramite OVERRIDE per ogni istanza (oggetto) della classe derivata **ResponsabileDiProgetto** aggiungendo il bonus

Linguaggio C++: polimorfismo – le funzioni virtuali

Le funzioni virtuali sono un altro meccanismo che ci permette di realizzare il **polimorfismo** con C++, una delle più importanti caratteristiche dei linguaggi orientati agli oggetti, che permette ad oggetti "simili" di rispondere in modo diverso agli stessi comandi

Una funzione virtuale è una funzione membro dichiarata come virtual in una classe base e ridefinita in una classe derivata.

Quando si eredita una classe contenente una funzione virtuale, la classe derivata ridefinisce la funzione virtuale secondo le proprie esigenze.

Il compito principale della funzione virtuale definita nella classe base è quello di definire appunto al forma dell'interfaccia della funzione.

Le ridefinizioni nelle classe derivate, invece, implementa le specifiche azioni relative alle situazioni gestite dalla classe derivata stessa.

Linguaggio C++: polimorfismo – le funzioni virtuali

Quando si accede alle funzioni virtuali, per mezzo dell'operatore . (punto), tali funzioni si comportano come qualsiasi altra funzione membro.

La potenzialità delle funzioni virtuali, viene sfruttata quando si accede loro tramite puntatore(mediante l'operatore ->). Questo tipo di accesso consente di realizzare il polimorfismo run-time.

Quando un puntatore base punta ad un oggetto derivato che contiene una funzione virtuale, il C++ determina quale funzione chiamare sulla base del tipo di oggetto puntato. Questa determinazione viene eseguita run-time.

Pertanto, al variare del tipo di oggetto derivato puntato, cambia anche la versione della funzione virtuale che verrà eseguita.

Linguaggio C++: polimorfismo – le funzioni virtuali

Prendiamo in considerazione un semplice esempio: **un insieme di classi che rappresentano dei veicoli a motore**: auto, moto, sidecar e cosi via

Tutti i veicoli fanno riferimento alla classe base **Veicolo**

Supponiamo che ogni classe sia dotata di un funzione **numRuote** che indica il numero di ruote. Indipendentemente dal veicolo, risulterebbe comodo trattare la funzione **numRuote** come un metodo della classe base **Veicolo**

In altre parole per sapere quante ruote possiede un certo veicolo non faremmo altro che richiamare la funzione **numRuote** della classe padre veicolo, sarà poi il compilatore a decidere in maniera automatica (**binding dinamico**) a quale classe derivata appartiene la funzione numRuote

Linguaggio C++: le funzioni virtuali (esempio)

```
#include <iostream>

using namespace std;


class Veicolo
{
public:
      virtual void numRuote() {};
};


class Automobile : public Veicolo
{
public:
      void numRuote();
};

void Automobile::numRuote()
{
cout << "automobile appartiene a veicolo ed ha 4 ruote" << endl;
}
```

Dichiariamo la funzione **numRuote** della classe base **Veicolo** come **funzione virtuale** e poi la implementiamo nelle classi derivate **Automobile**, **Moto** e **Sidecar**
Una funzione virtuale si dichiara precedendo il suo prototipo con la parola chiave **virtual** nella classe base
Deve essere dichiarata **OBBLIGATORIAMENTE** all'interno della classe base

OVERRIDING del metodo **numRuote**
Ciascuna classe derivata implementa il metodo virtuale definito nella classe base **Veicolo** secondo le sue esigenze

Linguaggio C++: le funzioni virtuali (esempio)

```
class Moto : public Veicolo
{
public:
      void numRuote();
};

void Moto::numRuote()
{
cout << "moto appartiene a veicolo ed ha 2 ruote" << endl;
};


class Sidecar : public Veicolo
{
public:
      void numRuote ();
};

void Sidecar::numRuote ()
{
cout << "sidecar appartiene a veicolo ed ha 3 ruote" << endl;
};
```

OVERRIDING del metodo **numRuote**
Ciascuna classe derivata implementa il metodo virtuale definito nella classe base **Veicolo** secondo le sue esigenze

Linguaggio C++: le funzioni virtuali (esempio)

Dietro una semplice codifica di questo tipo si nasconde tutta la potenza del polimorfismo: il sistema chiama in modo automatico il metodo **numRuote** dell'oggetto che è selezionato in quel momento, senza che ci si debba preoccupare se si tratti di automobile, moto o sidecar.

In altre parole, potremmo dire che il polimorfismo consente ad oggetti differenti (ma collegati tra loro) la flessibilità di rispondere in modo differente allo stesso tipo di messaggio

```
int main(int argc, char** argv)
{
Veicolo* VeicoliAMotore[3] = { new Automobile, new Moto, new Sidecar };

for(int i = 0; i < 3; i++)
   {
   VeicoliAMotore[i]->numRuote();
   }

return 0;
}
```

Usiamo l'allocazione dinamica (puntatori)

Quando viene ereditata una funzione virtuale viene ereditata anche la sua natura virtuale.
Questo significa che quando una classe derivata che abbia ereditato una funzione virtuale viene a sua volta utilizzata come classe base per un ulteriore classe derivata la funzione resta virtuale.

Linguaggio C++: esercizi extra da svolgere

Esercizio-1 classe **Rettangolo** con proprietà e funzioni membro **public**

Implementare in C++ la classe **Rettangolo** qui descritta attraverso il suo relativo diagramma delle classi UML **con il costruttore che di default costruisce rettangoli con base = 3 ed altezza = 4**

Rettangolo
+ base: **REAL** + altezza: **REAL**
+ **FUNZIONE** perimetro (**VAL** b : **REAL**, **VAL** h: **REAL**) : **REAL** + **FUNZIONE** area (**VAL** b : **REAL**, **VAL** h: **REAL**) : **REAL** + **FUNZIONE** diagonale (**VAL** b : **REAL**, **VAL** h: **REAL**) : **REAL** + **COSTRUTTORE** **Rettangolo** ( ) + **DISTRUTTORE** **~Rettangolo** ( )

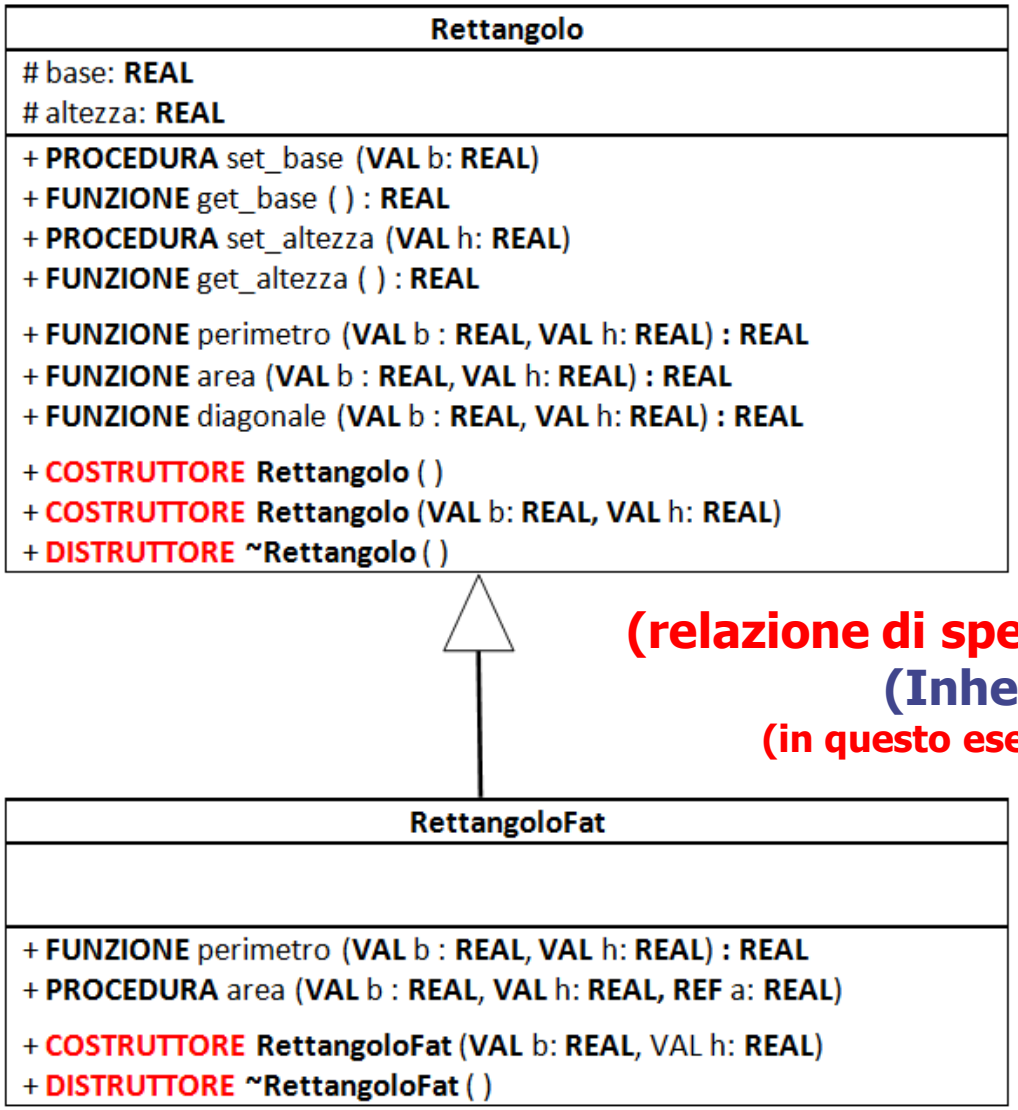
Linguaggio C++: esercizi extra da svolgere

Esercizio-2 classe **Rettangolo** con proprietà **private** e funzioni membro **public**

Implementare in C++ la classe **Rettangolo** qui descritta attraverso il suo relativo diagramma UML

Rettangolo
- base: **REAL** - altezza: **REAL**
+ **PROCEDURA** set_base (**VAL** b: **REAL**) + **FUNZIONE** get_base ( ) : **REAL** + **PROCEDURA** set_altezza (**VAL** h: **REAL**) + **FUNZIONE** get_altezza ( ) : **REAL** + **FUNZIONE** perimetro (**VAL** b : **REAL**, **VAL** h: **REAL**) **: REAL** + **FUNZIONE** area (**VAL** b : **REAL**, **VAL** h: **REAL**) **: REAL** + **FUNZIONE** diagonale (**VAL** b : **REAL**, **VAL** h: **REAL**) **: REAL** + **COSTRUTTORE Rettangolo** ( ) + **COSTRUTTORE Rettangolo** (**VAL** b: **REAL**, **VAL** h: **REAL**) + **DISTRUTTORE ~Rettangolo** ( )

Linguaggio C++: esercizi extra da svolgere



Esercizio-3 Rettangolo

Implementare in C++ **la classe base Rettangolo** e la **classe derivata RettangoloFat** qui descritta attraverso il seguente modello delle classi UML

(relazione di specializzazione-generalizzazione)
(Inheritance Relationship)
(in questo esempio EREDITARIETA' SEMPLICE)

Autore: Prof. Rio Chierego

Marzo 2016

Linguaggio C++: esercizi extra da svolgere

Esercizio-4 classe **Bicicletta** con proprietà **private** e funzioni membro **public**

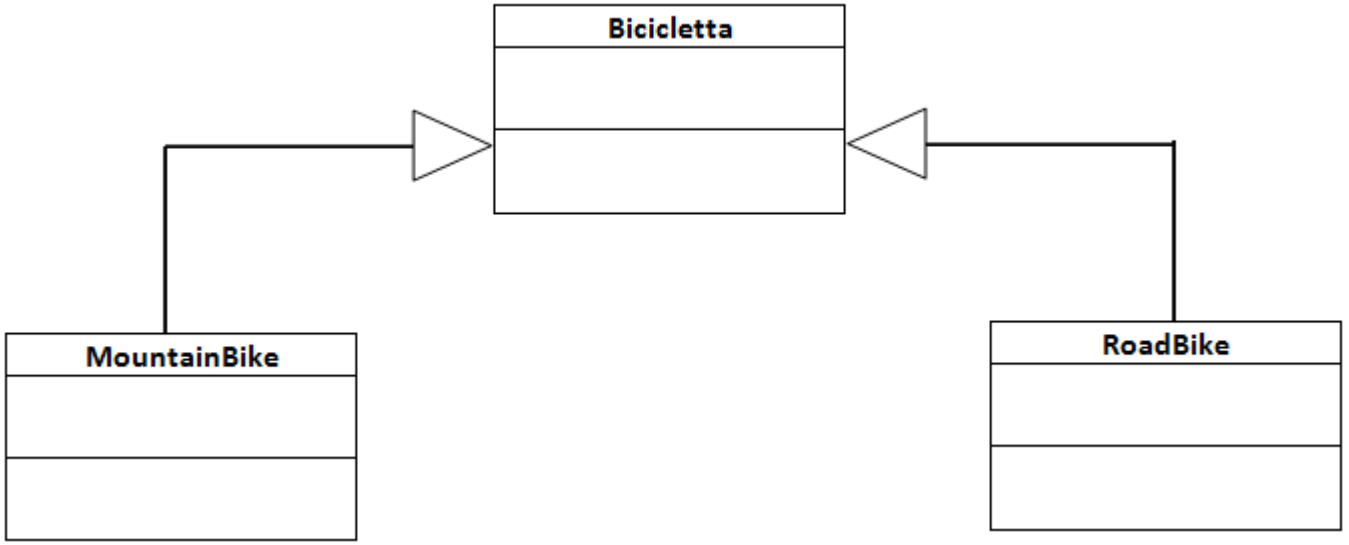
Implementare in C++ la classe **Bicicletta** qui descritta attraverso il seguente modello delle classi UML

Bicicletta
- cadenza: **INT** - marcia: **INT** - velocita: **INT**
+ **PROCEDURA** set_cadenza (**VAL** c: **INT**) + **FUNZIONE** get_cadenza ( ) : **INT** + **PROCEDURA** set_marcia (**VAL** m: **INT**) + **FUNZIONE** get_marcia ( ) : **INT** + **PROCEDURA** set_velocita (**VAL** v: **INT**) + **FUNZIONE** get_velocita ( ) : **INT** + **PROCEDURA** accelera (**VAL** inc : **INT**) + **PROCEDURA** frena (**VAL** dec : **INT**) + **PROCEDURA** mostra_stato ( ) + COSTRUTTORE **Bicicletta** ( ) + COSTRUTTORE **Bicicletta** (**VAL** c: **INT, VAL** m: **INT, VAL** v: **INT**) + DISTRUTTORE **~Bicicletta** ( )

Linguaggio C++: esercizi extra da svolgere

Esercizio-4 Bicicletta (EREDITARIETA' SEMPLICE)

Implementare in C++ la **classe base Bicicletta** supponendola come classe di generalizzazione delle classi derivate (classi di specializzazione) **MountainBike** e **RoadBike** secondo il seguente modello delle classi UML



(relazione di specializzazione-generalizzazione)
(Inheritance Relationship)
(in questo esempio EREDITARIETA' SEMPLICE)

Linguaggio C++: esercizi extra da svolgere

Esercizio-5 Bicicletta (EREDITARIETA' SEMPLICE)

Ecco la classe base **Bicicletta** descritta il seguente modello delle classi UML

Bicicletta
cadenza: **INT** # marcia: **INT** # velocita: **INT**
+ **PROCEDURA** set_cadenza (**VAL** c: **INT**) + **FUNZIONE** get_cadenza ( ) : **INT** + **PROCEDURA** set_marcia (**VAL** m: **INT**) + **FUNZIONE** get_marcia ( ) : **INT** + **PROCEDURA** set_velocita (**VAL** v: **INT**) + **FUNZIONE** get_velocita ( ) : **INT** + **PROCEDURA** accelera (**VAL** inc : **INT**) + **PROCEDURA** frena (**VAL** dec : **INT**) + **PROCEDURA** mostra_stato ( ) + COSTRUTTORE **Bicicletta** ( ) + COSTRUTTORE **Bicicletta** (**VAL** c: **INT**, **VAL** m: **INT**, **VAL** v: **INT**) + DISTRUTTORE **~Bicicletta** ( )

Linguaggio C++: esercizi extra da svolgere

Esercizio-5 Bicicletta (EREDITARIETA' SEMPLICE)

Ecco la classe base **MountainBike** descritta attraverso il seguente modello delle classi UML

MountainBike
- num_a: **INT** - tipo_a: **STRING**
+ **PROCEDURA** set_num_a (**VAL** n: **INT**) + **FUNZIONE** get_tipo_a ( ) : **INT** + **PROCEDURA** set_tipo_a (**VAL** t: **STRING**) + **FUNZIONE** get_tipo_a ( ) : **STRING** + **PROCEDURA** set_velocita (**VAL** v: **INT**) + **FUNZIONE** get_velocita ( ) : **INT** + **PROCEDURA** accelera (**VAL** inc : **INT, VAL** attr: **INT**) + **PROCEDURA** mostra_stato ( ) + COSTRUTTORE **MountainBike** (**VAL** c: **INT, VAL** m: **INT, VAL** v: **INT, VAL** n: **INT, VAL** t: **STRING**) + DISTRUTTORE **~MountainBike** ( )

Linguaggio C++: esercizi extra da svolgere

Esercizio-5 Bicicletta (EREDITARIETA' SEMPLICE)

Ecco la classe base **RoadBike** descritta attraverso il seguente modello delle classi UML

RoadBike
- profilo: **STRING** - cambio: **STRING**
+ **PROCEDURA** set_profilo (**VAL** p: **STRING**) + **FUNZIONE** get_profilo ( ) : **STRING** + **PROCEDURA** set_cambio (**VAL** t: **STRING**) + **FUNZIONE** get_cambio ( ) : **STRING** + **PROCEDURA** mostra_stato ( ) + COSTRUTTORE **RoadBike** (**VAL** c: **INT, VAL** m: **INT, VAL** v: **INT, VAL** p: **STRING, VAL** t: **STRING**) + DISTRUTTORE **~RoadBike** ( )

Linguaggio C++: esercizi extra da svolgere

Esercizio-6 Crono

Ecco la classe **Crono** descritta attraverso il seguente modello delle classi UML

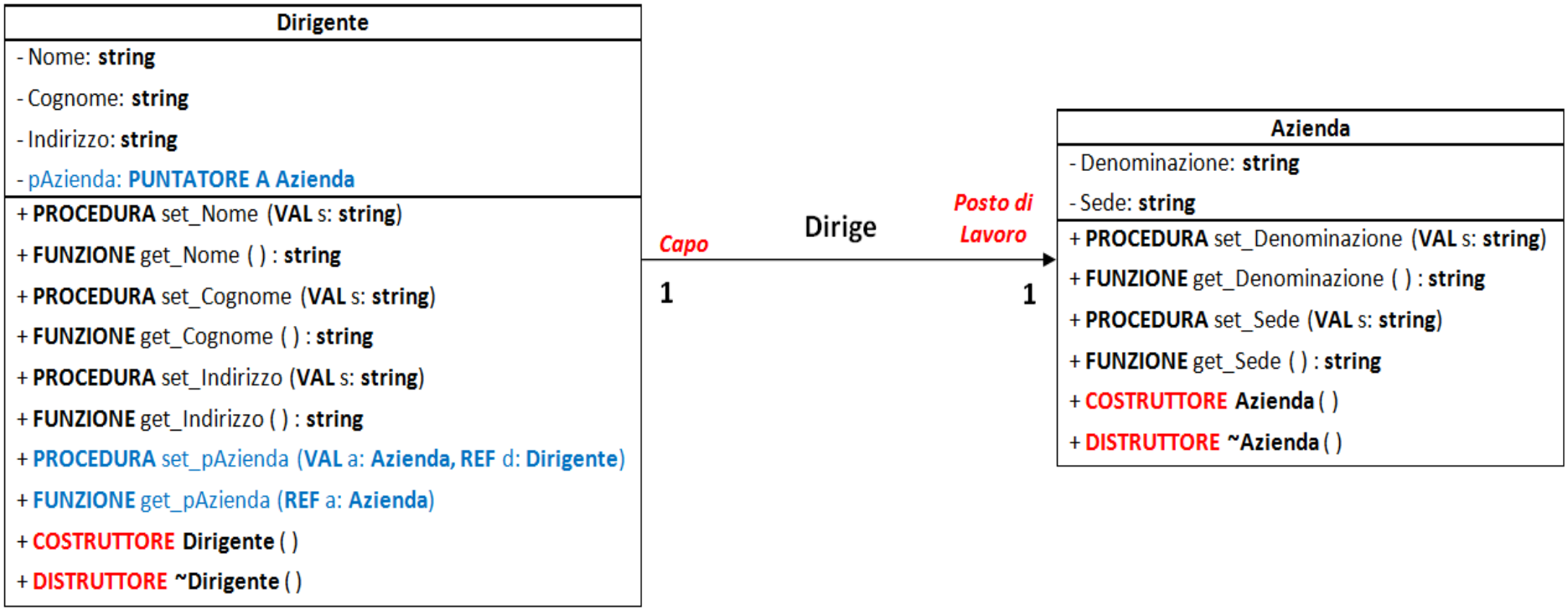
Crono
- tf: **INT** - ti: **INT**
+ **PROCEDURA** set_ti (**VAL** t1: **INT**) + **FUNZIONE** get_ti ( ) : **INT** + **PROCEDURA** set_tf (**VAL** t2: **INT**) + **FUNZIONE** get_tf ( ) : **INT** + **PROCEDURA** go ( ) + **PROCEDURA** stop ( ) + **FUNZIONE** print ( ) : **INT** + COSTRUTTORE **Crono** () + DISTRUTTORE **~Crono** ( )

Occorre utilizzare le due variabili
ti = tempo iniziale
tf = tempo finale
che verranno registrate tramite la funzione **time(0)** che fornisce l'ora attuale espressa in secondi.
Sfrutteremola possibilità
che la differenza **tf - ti** viene espressa in secondi.
Il menù del Crono avra' le seguenti voci:
g) Go (fa partire il conteggio inizializzando **ti**)
s) Stop (ferma il conteggio inizializzando **tf**)
p) Print (Visualizza il tempo **tf - ti**)
x) Exit (Esce dall'applicazione)

Linguaggio C++: esercizi extra da svolgere

Esercizio-7 Dirigente-Azienda

Ecco la relazione di **associazione (Use Relationship) unidirezionale** tra le classi **Dirigente** ed **Azienda** descritta attraverso il seguente modello delle classi

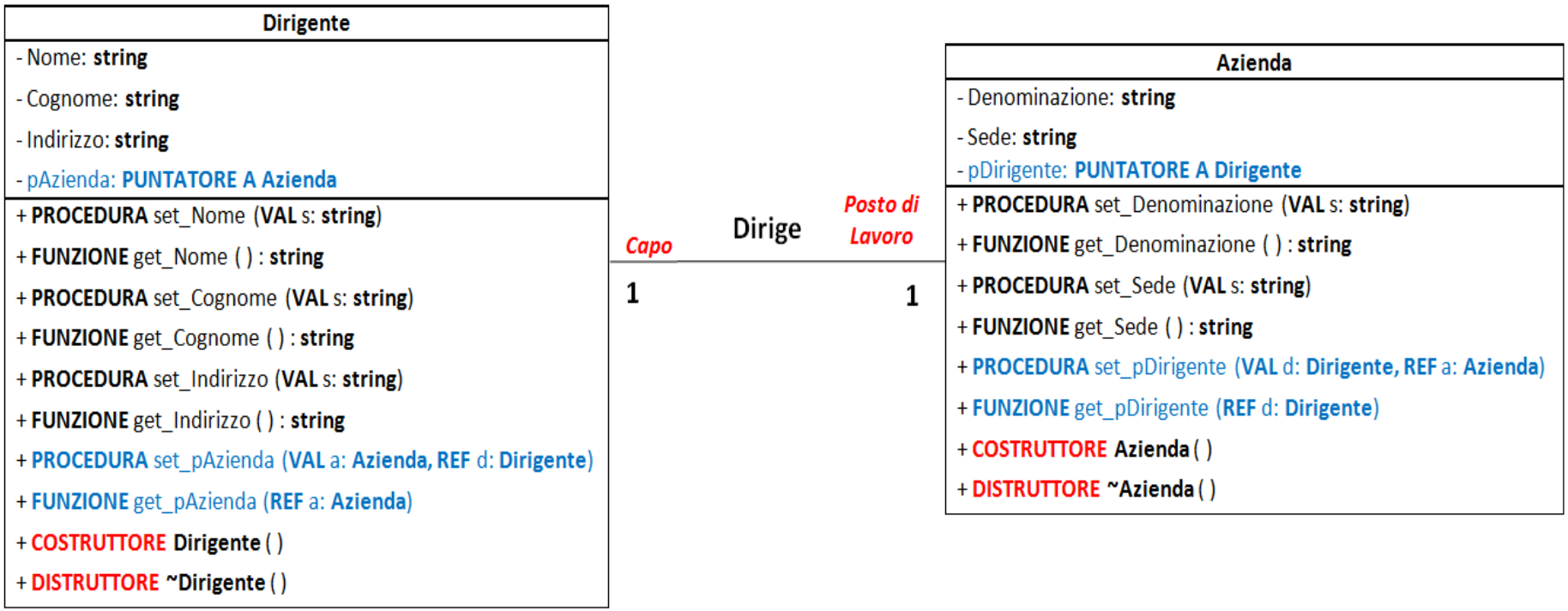


Dopo avere implementato in C++ le due classi **Dirigente** ed **Azienda** e realizzato **l'associazione** tra di esse di **molteplicità 1:1**, si istanzi un dirigente e l'azienda da lui diretta

Linguaggio C++: esercizi extra da svolgere

Esercizio-8 Dirigente-Azienda

Ecco la relazione di **associazione (Use Relationship) bidirezionale** tra le classi **Dirigente** ed **Azienda** descritta attraverso il seguente modello delle classi UML



Dopo avere implementato in C++ le due classi **Dirigente** ed **Azienda** e realizzato **l'associazione** tra di esse di **molteplicità 1:1**, si istanzi un dirigente e l'azienda da lui diretta e poi il viceversa

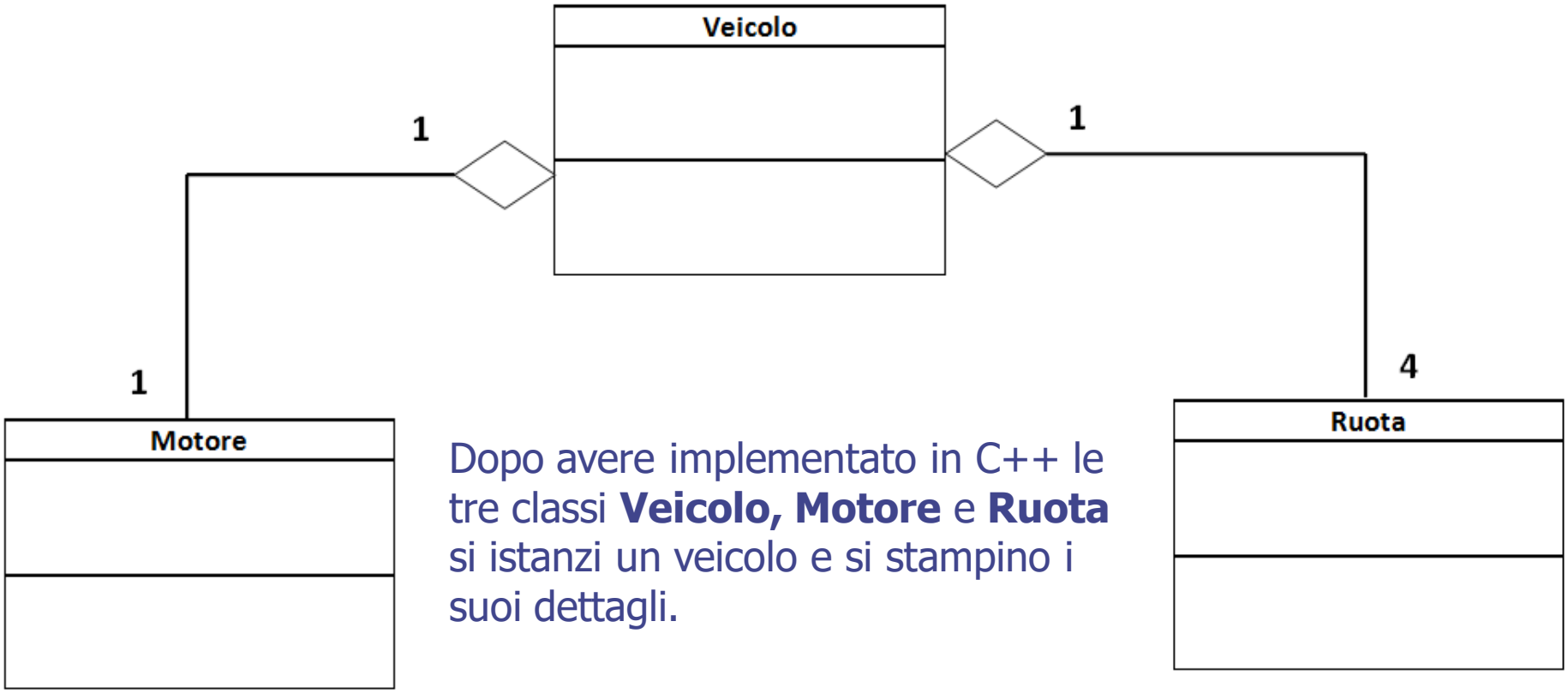
Autore: Prof. Rio Chierego

Marzo 2016

Linguaggio C++: esercizi extra da svolgere

Esercizio-9 Aggregazione lasca Veicolo-Motore-Ruota

Ecco la relazione di **aggregazione (Containment Relationship) lasca (o debole)** tra le classi **Veicolo** (**classe contenitore**) e le classi **Motore** e **Ruota** (**classi componenti**) descritta attraverso il seguente modello delle classi UML



Dopo avere implementato in C++ le tre classi **Veicolo, Motore** e **Ruota** si istanzi un veicolo e si stampino i suoi dettagli.

Linguaggio C++: esercizi extra da svolgere

Esercizio-9 Aggregazione lasca Veicolo-Motore-Ruota

Tale relazione si realizza in C++ introducendo nella classe contenitore uno o più puntatori agli oggetti contenuti. Il costruttore riceve in ingresso il/i puntatore/i all'oggetto/i contenuto/i

Veicolo
- Marca: **string** - Modello: **string** - Costo: **FLOAT** - pMotore: **PUNTATORE A Motore** - pRuota1: **PUNTATORE A Ruota** - pRuota2: **PUNTATORE A Ruota** - pRuota3: **PUNTATORE A Ruota** - pRuota4: **PUNTATORE A Ruota**
+ **PROCEDURA** set_Marca (**VAL** s: **string**) + **FUNZIONE** get_Marca ( ) : **string** + **PROCEDURA** set_Modello (**VAL** s: **string**) + **FUNZIONE** get_Modello ( ) : **string** + **PROCEDURA** set_Costo (**VAL** c: **FLOAT**) + **FUNZIONE** get_Costo ( ) : **FLOAT** + **PROCEDURA** datiVeicolo () + **COSTRUTTORE Veicolo** (**REF** m: **Motore**, **REF** r1:**Ruota**, **REF** r2: **Ruota**, **REF** r3: **Ruota**, **REF** r4: **Ruota**) + **DISTRUTTORE ~Veicolo** ( )

Linguaggio C++: esercizi extra da svolgere

Esercizio-9 Aggregazione lasca Veicolo-Motore-Ruota

Motore
- Cavalli: **INT**
- Pistoni: **INT**
+ **PROCEDURA** set_Cavalli (**VAL** c: **INT**)
+ **FUNZIONE** get_Cavalli ( ) : **INT**
+ **PROCEDURA** set_Pistoni (**VAL** p: **INT**)
+ **FUNZIONE** get_Pistoni ( ) : **INT**
+ **PROCEDURA** datiMotore ()
+ **COSTRUTTORE** **Motore** ( )
+ **DISTRUTTORE** **~Motore** ( )

Ruota
- Diametro: **FLOAT**
- Battistrada: **FLOAT**
+ **PROCEDURA** set_Diametro (**VAL** d: **FLOAT**)
+ **FUNZIONE** get_Diametro ( ) : **FLOAT**
+ **PROCEDURA** set_Battistrada (**VAL** b: **FLOAT**)
+ **FUNZIONE** get_Battistrada ( ) : **FLOAT**
+ **PROCEDURA** datiRuota ()
+ **COSTRUTTORE** **Ruota** ( )
+ **DISTRUTTORE** **~Ruota** ( )