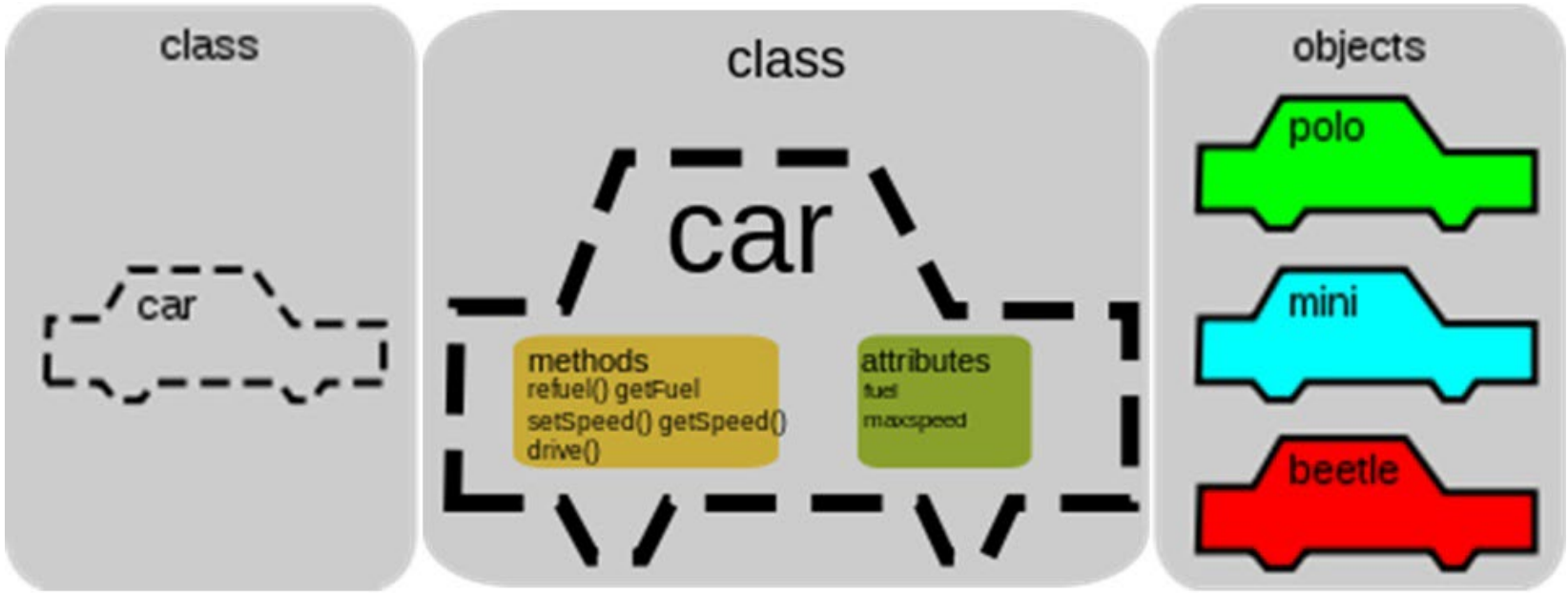
Il linguaggio C++



Principi di OOP nel linguaggio C++

Autore: Prof. Rio Chierego

23/04/2025

Aprile 2025

Sommario

- **Linguaggio C++: breve storia**
- **Peculiarità del linguaggio C++ rispetto al C**
  - il tipo **bool,** i tipi **struct, union e enum**
  - il flusso **iostream**
  - l'operatore di **scope resolution ::**
  - il qualificatore **const**
  - il **linkage**
  - il **namespace**
  - la direttiva **using**
  - le funzioni con **argomenti di default**
  - l'**overloading** funzioni
  - le variabili **reference**
  - **-** la classe **string** ed i suoi metodi principali
  - **-** la classe **vector** ed i suoi metodi principali
  - **-** la classe **list** ed i suoi metodi principali
- **L'applicazione dei principi base dell'OOP programming nel linguaggio C++**
  - il puntatore **this**
  - **-** le **funzioni virtuali**
- **Esercizi svolti, da svolgere ed EXTRA**

Linguaggio C++: PREMESSA

Queste slide non intendono in alcun modo essere un corso esaustivo dell'intera sintassi e di tutte le potenzialità possedute dal **linguaggio C++**.

Esse si pongono l'obiettivo di accompagnare e sostenere lo studente con una buona conoscenza della programmazione imperativa esercitata attraverso il **linguaggio C** e che intenda ora approcciarsi alla programmazione ad oggetti partendo dall'indubbio vantaggio dell'esistenza di una base comune ai due linguaggi di programmazione di alto livello.

Di conseguenza le slide cercano di porre l'attenzione soprattutto sulle differenze fondamentali e su alcune caratteristiche specifiche che il **linguaggio C++** possiede rispetto al suo linguaggio "progenitore" mettendole in evidenza, ogni volta che sarà possibile, con esercizi ed esempi ad hoc, molti dei quali completamente svolti

Tutti gli esercizi di esempio svolti sono presenti nel file zippato
CHIEREGO-OOP-Programming-Linguaggio-CPP.zip

Autore: Prof. Rio Chierego

Aprile 2025

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 all'interno del codice C++

```c
#include <iostream>

//Indispensabile per le funzioni sulle stringhe
#include <string.h>
//Indispensabile per le funzioni matematiche
#include <math.h>
//Indispensabile per le funzioni di allocazione
#include <malloc.h>

int main(int argc, char** argv)
{
char s[30 + 1];
int a;
int *p;
float x = 4.00, y;

//Lettura e scrittura sile C
printf("Inserisci valore: ");
scanf("%d", &a);

//Copia in s "Hello!"
strcpy(s, "Hello!");

//Confronta "Hello" con "Paperino"
strcmp(s, "Paperino");

printf("Lunghezza stringa %s = %d\n", s, strlen(s));

y = sqrt(x);
printf("Radice quadrata di %.2f = %.2f\n", y, x);

p = (int*) malloc (sizeof(int));
return 0;
}
```

N.B

Ovviamente, per garantire la compatibilità all'indietro, è possibile utilizzare all'interno di un progetto DEV-CPP per il linguaggio C++, tutte le istruzioni e le funzioni illustrate per linguaggio C

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.** Avranno dunque senso in C++ istruzioni del tipo:

```
.....
bool trovato;                  //dichiarazione di una variabile boooleana
......
trovato = false;               //inizializzazione di una variabile boooleana
......
if (trovato == true)           //condizione logica con una variabile boooleana
......
```

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 Cittadino
 {
  char Cognome[30 + 1];
  char Nome[30 + 1];
  };
Cittadino citt;      //definizione di una variabile di tipo "Cittadino"
```

Linguaggio C++: peculiarità – iostream

Stream standard di input e di output: cin e cout

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

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

Esempio

```
#include <iostream>
int main(int argc, char** argv) {
int a = 2;
std::cout << "Il linguaggio C++" << std::endl;
std::cout << "Il valore di a e': " << a  << std::endl << std::endl;
return 0;
 }
```

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

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

Esempio

1. vedi Celsius.dev

```
#include <iostream>
int main(int argc, char** argv) {
float temp;
std::cout << "Temperatura in gradi Celsius"<< std::endl;
std::cin >> temp;
std::cout << "Temperatura in gradi Fahrenheit : "<<  32 + temp * 9/5 << std::endl;
return 0;
 }
```

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

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**

2. vedi ScopeRes.dev

```
#include <iostream>
int x = 0;          //variabile globale inizializzata a 0
int main(int argc, char** argv)
{
int x = 5;          //variabile locale omonima inizializzata a 5
::x = 4;            // Assegna alla variabile globale omonima il valore 4
std::cout <<  x << std::endl;        //verrà mostrato a video il valore 5
std::cout << ::x << std::endl;       //verrà mostrato a video il valore 4
return 0;
}
```

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 (ossia "sola lettura")

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

Ovviamente l'istruzione

DIMENSIONE++;

darebbe un errore di compilazione dicendo che si sta tentando di modificare una variabile di "sola lettura"

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>
#include "name1.h"
#include "name2.h"
 int main(int argc, char** argv)
{
std::cout << "Somma = " << primo::x + secondo::x << std::endl;
return 0;
}
```

3. vedi Namespace.dev

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>
 int main(int argc, char** argv)
{
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::endl;
 int main(int argc, char** argv)
{
cout << "Il linguaggio C++ standard" << endl;
return 0;
}
```

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;
 int main(int argc, char** argv)
{
cout << "Il linguaggio C++ standard" << endl;
return 0;
}
```

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 (**11.2**); // il compilatore assumerà la chiamata **ordine (11, 7.1);**
// e non come si poteva credere erroneamente **ordine (8, 11.2);**

ordine (**21, 56.1**); // il compilatore assumerà la chiamata **ordine (21, 56.1);**

N.B. quest'ultima chiamata è perfettamente equivalente al C

Fare attenzione:

4. vedi Default.dev

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 – l'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:

FUNZIONI	**PROCEDURE**
int **minimo** (int x, int y);	ma anche void **minimo** (int x, int y, int* z)
float **minimo** (float x, float y);	ma anche void **minimo** (float x, float y, float* z)

N.B. Anche nel linguaggio C++ (come nel linguaggio C) i sottoprogrammi si chiamano sempre "funzioni".

Linguaggio C++: peculiarità – l'overloading delle funzioni

5. vedi Minimo.dev

```
#include <iostream>

using namespace std;

//OVERLOADIN FUNZIONE minimo
void minimo (int, int, int*);
void minimo (float, float, float*);

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

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

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

cout << "\nPrimo numero reale : ";
cin >> a;
cout << "Secondo numero reale : ";
cin >> b;

cout << endl << "Funzione minimo (tra REALI) con passaggio parametro x INDIRIZZO";
minimo (a, b, &m1);   //terzo parametro passato per indirizzo
cout << endl << "Il minimo tra " << a << " e " << b << " e' = " << m1;

return (0);
}
```

Ho scelto di implementare solo le due PROCEDURE

Linguaggio C++: peculiarità – l'overloading delle funzioni

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

return;
}

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

return;
}
```

L'overloading delle funzioni: esercizi proposti

1) Effettuare l'overloading della funzione **Load_v** che effettua **la lettura** degli elementi di un vettore monodimensionale di interi
2) Effettuare l'overloading della funzione **Print_v** che effettua **la stampa** degli elementi di un vettore monodimensionale di interi
3) Effettuare l'overloading della funzione **Sort_v** che effettua **l'ordinamento** di un vettore monodimensionale di interi
4) Effettuare l'overloading della funzione **Search_v** che effettua la **ricerca** di un elemento in un un vettore monodimensionale di interi
5) Effettuare l'overloading della funzione **Rotate_v** che effettua le due possibili **rotazioni** (a dx ed a sx) degli elementi di un vettore monodimensionale di interi
6) Effettuare l'overloading della funzione **MaxMin_v** che individua il **massimo** ed il **minimo** in un vettore monodimensionale di interi
7) Effettuare l'overloading della funzione **Media_v** che individua la **media** degli elementi un vettore monodimensionale di interi
8) Effettuare l'overloading della funzione **Diagonal_v** che effettua **la stampa** degli elementi della diagonale principale di una marice quadrata

Linguaggio C++: peculiarità – variabili 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 un sottoprogramma (procedura o funzione):

- **per valore (BY VALUE)**: la funzione riceve una copia degli argomenti (parametri attuali)
- **per riferimento (BY REFERENCE)**: la funzione riceve una copia dell'indirizzo degli argomenti (parametri attuali)

Utilizzando **il linguaggio C** abbiamo visto che, in mancanza delle variabili reference, tale passaggio dei parameri necessita dell'utilizzo di variabili speciali chiamate **puntatori**, con operatori dedicati (operatore indirizzo **&** ed operatore contenuto *):

Il linguaggio C++, oltre a possedere le variabili di tipo puntatore (e quindi potendo implementare il passaggio dei parametri per indirizzo come il C) permette il **passaggio dei parametri per riferimento (BY REFERENCE)** anche utilizzando **le variabili reference.**

Una variabile reference è sempre un contenitore di indirizzi (come un puntatore), ma si può adoperare come una normale variabile, senza doversi preoccupare degli peratori propri dei puntatori (& e *).

Il nuovo costrutto richiede l'uso dell'operatore **& (ampersand) in fase di dichiarazione** che non ha più il "vecchio" significato di **"indirizzo di"** ma quello di **"fa 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 – variabili reference

6. vedi Reference.dev

```
#include <iostream>
using namespace std;
int main(int argc, char** argv)
{
int dato = 5;                 // N.B. In fase di DICHIARAZIONE è OBBLIGATORIA
                              // l'inizializzazione di una variabile di tipo reference
int& refdato = dato;          // da questo punto in poi essa è un ALIAS della variabile dato
```

L'output prodotto dal seguente eseguibile sarà il seguente

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

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

return 0;
}
```

```
6
6
7
7
0x9ffe44
0x9ffe44
```

Linguaggio C++: polimorfismo – l'OVERLOADING e reference

7. vedi Moltiplica.dev

```
#include <iostream>

using namespace std;

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

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

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

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

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

return (0);
}
```

Si noti che il tipo int* è diverso dal tipo int&
(N.B. non sarebbe stato così tra int e int&)

Linguaggio C++: polimorfismo – l'OVERLOADING e reference

```
// Funzione moltiplica passaggio parametro x INDIRIZZO
void moltiplica (int a, int b, int* c)
{
*c = a * b;
return;
}

// Funzione moltiplica passaggio parametro x REFERENCE
void moltiplica (int a, int b, int& c)
{
c = a * b;
return;
}
```

Si noti che il tipo int* è diverso dal tipo int&
(non sarebbe stato così tra int e int&)

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 – 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>
#include <string>
using namespace std;

int main()
{
string parola;

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

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.

La funzione **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 della classe **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
```

8. vedi Stringhe-Premessa.dev

N.B. Con le stringhe in stile C++ la quantità di memoria allocata viene adattata automaticamente al numero di caratteri posseduti, superando così il problema del dimensionamento su un numero fissato di caratteri di una stringa in stile C.

RICORDA però che….
il linguaggio C++, comunque, mantiene la gestione delle stringhe in stile C ed anzi esse sono ancora fondamentali, soprattutto in alcuni contesti di programmazione in cui al contrario si ha l'esigenza di lavorare con variabili stringa a dimensione fissa, quali sono appunto le stringhe in stile C.

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 << endl;
```

  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<< endl;

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

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

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

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

9. vedi Stringhe-Metodi-1.dev

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 " << endl;
```

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

- **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)<< endl;

// visualizza "rino"
cout<<parola.substr(2,4)<< endl;

// visualizza "rino"
cout<<parola.substr(2,8)<<endl;
```

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

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char** argv)
{
string parola = "Torino";
string esito;

//Viene visualizzato "rin" perchè inizio ok & len ok
esito = parola.substr (2, 3 );
cout << esito << endl;

//Viene visualizzato "ino" perchè inizio ok & len not ok
esito = parola.substr (3, 4 );
cout << esito << endl;

//ERRORE: inizio not ok & len ok
esito = parola.substr (-1, 4);
cout << esito << endl;
```

Errore ottenuto a run-time

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::substr: __pos (which is 18446744073709551615) > this->size() (which is 6)
```

```
return 0;
}
```

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

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char** argv)
{
string parola = "Napoli";
string esito;

//Viene visualizzato "apol" perchè inizio ok & len ok
esito = parola.substr (1, 4 );
cout << esito << endl;

//Viene visualizzato "p" perchè inizio ok & len not ok
esito = parola.substr (2, 1 );
cout << esito << endl;

//ERRORE: inizio not ok & len not ok
esito = parola.substr (8, 4);
cout << esito << endl;
```

Errore ottenuto a run-time

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::substr: __pos (which is 8) > this->size() (which is 6)
```

```
return 0;
}
```

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";

//Viene visualizzato "Milano" perchè inizio ok & len ok & str ok
cout << parola.replace (0, 4, "Mila") << endl;

//Viene visualizzato "Merano" perchè inizio ok & len ok & str ok
cout << parola.replace (1, 2, "er") << endl;

//Viene visualizzato "Loano" anche se inizio ok & len ok > str (ok)
cout << parola.replace (0, 3, "Lo") << endl;

//Viene visualizzato "Loreto" anche se inizio ok & len ok < str (ok)
cout << parola.replace (2, 3, "reto") << endl;

//ERRORE: inizio not ok & len ok > str (ok)
cout << parola.replace (7, 3, "irat") << endl;
```

Errore ottenuto a run-time

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::replace: __pos (which is 7) > this->size() (which is 6)
```

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:

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

for (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:

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

for (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++: peculiarità – la classe **string**

L'**inserimento** di una stringa in un'altra può essere realizzato con il metodo

insert (int start, string s**)**

- start è la posizione da cui inizia l'inserimento; le posizioni si iniziano a contare da zero;
- s è la stringa da inserire a partire dalla posizione *start*

Esempi di utilizzo

1)
```
...
string parola = "Torino";
// Verrà modificata la parola iniziale e mostrato
// il suo nuovo valore ossia  "Tortellino"
cout << parola.insert(3, "tell") << endl;
```

2)
```
...
string parola = "Torino";
// Verrà mostrato un messaggio di errore perchè abbiamo
// sfondato il limite per l'indice che va da 0 a 6 (TAPPO)
cout << parola.insert(7, "tell") << endl;
```
```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::insert: __pos (which is 7) > this->size() (which is 6)
```

3)
```
...
string parola = "Torino";
// Verrà mostrato un messaggio di errore perchè abbiamo
// sfondato il limite per l'indice che va da 0 a 6 (TAPPO)
cout << parola.insert(-1, "tell") << endl;
```
```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::insert: __pos (which is 18446744073709551615) > this->size() (which is 6)
```

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

La **cancellazione** di una sotto-stringa all'interno di una stringa assegnata viene fatta tramite il metodo

erase (int start, int n)

- start è la posizione di inizio della cancellazione; le posizioni si iniziano a contare da zero;
- n (opzionale) è il numero di caratteri da cancellare; se non specificato cancella fino all'ultimo carattere;

Esempi di utilizzo

1)
```
...
string parola = "Torino";
// Così verrà mostrata la stringa "Tori" ...
cout << parola.erase(4, 2) << endl;
// ma anche così...
cout << parola.erase(4) << endl;
// ma anche così...
cout << parola.erase(4, 10) << endl;
```

2)
```
...
string parola = "Torino";
// Così verrà mostrata la stringa "Tino" ...
cout << parola.erase(1, 2) << endl;
```

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

Esempi di utilizzo

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char** argv)
{
string parola = "Napoli";

//Viene visualizzato ancora "Napoli" nonostante posizione 6
cout << parola.erase (6, 2) << endl;

//ERRORE -- sfondamento a destra
cout << parola.erase (7, 2) << endl;
```

Errore ottenuto a run-time

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::erase: __pos (which is 7) > this->size() (which is 6)
```

```
//ERRORE -- sfondamento a sinistra
cout << parola.erase (-1, 3) << endl;
```

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::erase: __pos (which is 18446744073709551615) > this->size() (which is 6)
```

Errore ottenuto a run-time

```
return 0;
}
```

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

Si può **ricercare** una sotto-stringa all'interno di una stringa assegnata tramite il metodo

find (string s, int start)

- s è la sotto-stringa da cercare;
- start è la posizione dalla quale iniziare la ricerca;

N.B. Il metodo restituisce:

- Se TROVATA: la posizione all'interno della stringa s assegnata del primo carattere della sotto-stringa, contata a partire da zero;
- Se NON TROVATA: il valore -1

Esempi di utilizzo

1)
```
...
string parola = "Torino";
// Ricerca dall'inizio....verrà mostrato 2
cout << parola.find("ri", 0) << endl;
```

2)
```
...
string parola = "Torino";
// Ricerca dall'inizio....verrà mostrato -1
cout << parola.find("ra", 0) << endl;
```

3)
```
...
string parola = "Torino";
// Ricerca dall'inizio....verrà mostrato -1
cout << parola.find("ri", 3) << endl;
```

4)
```
...
string parola = "Toririno";
// In caso di ripetizioni verrà mostrata la posizione della prima occorrenza ossia 2
cout << parola.find("ri", 0) << endl;
```

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

Si può **ricercare** una sotto-stringa all'interno di una stringa assegnata tramite il metodo

find (string s, int start**)**

- s è la sotto-stringa da cercare;
- start è la posizione dalla quale iniziare la ricerca;

N.B. Il metodo restituisce:

- Se TROVATA: la posizione all'interno della stringa s assegnata del primo carattere della sotto-stringa, contata a partire da zero;
- Se NON TROVATA: il valore -1

Esempi di utilizzo

```
//ERRORE -- al contrario di erase (sfondamento a dx)
cout << parola.find ("Rio", 6) << endl;

//ERRORE -- sfondamento a sinistra
cout << parola.find ("Na", -1) << endl;

//ERRORE -- sfondamento a destra
cout << parola.find ("Na", -1) << endl;
```

Tutti errori ottenuto a run-time

```
18446744073709551615
```

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

```cpp
#include <iostream>
#include <string>
#include <string.h>          //libreria C

using namespace std;

int main(int argc, char** argv)
{
char s2[20];
string s="Tanto va che ci lascia lo zampino!";
string s1="la gatta al lardo ";

s.insert(9,s1);     //s è "Tanto va la gatta al lardo che ci lascia lo zampino!"
cout << s << endl;

cout << s.substr(9, 8) << endl; //stampa a video: "la gatta"

cout << s.erase(5, 38) << endl; //stampa a video: "Tanto zampino!"

cout << s.replace(6, 8, "vale!") << endl; //stampa a video: "Tanto vale!"

cout << s.find("a", 5) << endl; //stampa a video: 7

strcpy(s2, s.c_str());        //converte s in stile C e la copia in s2

cout << s2 << endl;           //stampa la stringa stile C    s2 "tanto vale!"

return 0;
}
```

10. vedi Stringhe-Metodi-2.dev

Linguaggio C++: peculiarità – Esercizio 1

1) Usando il C++, realizza un progetto DEV-CPP che svolga le istruzioni relative ai seguenti commenti:

```
//Definizione string parola1 con inizializzazione DI COPIA dal valore "E Giacomino"
.....
//Definizione di parola2 con inizializzazione DIRETTA a partire da parola1
.....
//Definizione di parola3 con inizializzazione DIRETTA con valore "si sposa!"
.....
//Concatenazione per creare con inizializzazione di COPIA stringa parola4 con valore
//"E Giacomino si sposa!"
.....
//Visualizzazione parola4
.....
//Visualizzare da parola4 della sottostringa "comi"
.....
//Visualizzare da parola4 la sottostringa = "sposi" utilizzando anche la concatenazione
.....
//Costruire da parola4 con inizializzione di COPIA la string parola5 = "sposini"
.....
//Trasformare parola5 ossia "sposini" in "susini"
.....
//Trasformare parola5 ossia "susini" in "sushi"
.....
```

Linguaggio C++: peculiarità – Esercizio 2

2) Usando il C++, realizza un progetto DEV-CPP che svolga le istruzioni relative ai seguenti commenti:

//Definizione con inizializzazione DI COPIA di string parola1 con valore "Forza Napoli sempre!"

.....

//Visualizzazione parola1

.....

//Costruzione stringa parola2 con inizializzazione DIRETTA a partire da parola1 usando il valore "Napoli"

.....

//Visualizzazione parola2

.....

//Accesso DIRETTO ai singoli caratteri di parola2 con metodo length() o size() a scelta

.....

//Accesso con METODO at ai singoli caratteri di parola2 (usare contatore i)

.....

//Conversione stringa parola2 in un array di char stile C con copia in variabile s (uso strcpy)

.....

//Visualizzazione della C-string s ricavata da parola2 con utilizzo puts()

.....

//Stampa dei caratteri presenti in s con ciclo while con criterio di arresto su tappo '\0' (usare contatore i)

.....

//Trasformazione con insert di parola 1 in "Forza Napoli Campione sempre!"

.....

//Trasformazione di parola1 con erase in "Forza sempre!"

.....

//Cercare se sottotringa "Napoli" è presente in parola1 (usare variabile int esito)

.....

//Cercare se sottostringa "za" è presente in parola1 (usare variabile int esito)

Linguaggio C++: peculiarità – Altri possibili esercizi

3. A. Leggi una stringa e trasformane le lettere minuscole in maiuscole.

   B. Leggi una stringa e trasformane le lettere minuscole in maiuscole e viceversa.

4. Leggi una stringa e determina quanto è lunga.

5. Leggi una stringa e un carattere e conta quante volte quel carattere è contenuto nella stringa.

6. Leggi una stringa e determina quante vocali contiene.

7. Leggi due stringhe e verifica quale è più lunga.

8. Leggi due stringhe e stampa per quante lettere fanno rima.

9. Leggi una stringa e verifica se è palindroma.

10. Leggi una stringa e verifica se contiene doppie.

11. Leggi una stringa e verifica che non ci siano caratteri ripetuti in essa.

12. Leggi una stringa e stampa qual è il carattere ripetuto più volte all'interno della stringa.

13. Date due stringhe verificare quanti caratteri hanno in comune, se un carattere compare due volte in entrambe le stringhe lo si conti due volte.

14. Date due stringhe verificare quanti caratteri diversi tra loro hanno in comune, se un carattere compare due volte in entrambe le stringhe lo si conti una volta.

15. Leggere una stringa e contare quanto caratteri diversi tra loro contiene.

16. Leggi due stringhe e verifica se una è una sottostringa dell'altra (ovvero se è contenuta nell'altra).

17. Leggi una stringa e verifica che sia composta solo da caratteri che compaiono più volte.

Linguaggio C++: peculiarità – la Standard Template Library

La **Standard Template Library** (STL) è una libreria software per il linguaggio di programmazione C++ che definisce quattro componenti principali: **contenitori**, **iteratori**, **algoritmi** e **funtori**.

La **STL** offre un insieme di classi C++. quali ad esempio i **contenitori** che possono essere usati con qualunque tipo di dato - sia esso predefinito o costruito dall'utente - che supporti alcune istruzioni elementari (copia, assegnazione, ecc.).

La **STL** è basata sui template, un approccio che permette il polimorfismo in fase di compilazione, nettamente più efficiente del polimorfismo in fase di esecuzione.

La **STL** fu la prima libreria di algoritmi e strutture dati generiche per il C++; si basa su quattro idee di fondo: programmazione generica, astrazione senza perdita di efficienza, modello di elaborazione di Von Neumann e semantica dei valori.

La **STL** è stata progettata e sviluppata presso la Hewlett-Packard da Alexander Stepanov e Meng Lee e sono state incluse nello standard ANSI/ISO nel 1995.

La **STL** e le idee contenute in essa, hanno avuto una notevole influenza nello sviluppo della C++ Standard Library con numerosi programmatori che hanno contribuito allo sviluppo di entrambe le librerie, malgrado ciò le due librerie sono rimaste distinte e nessuna delle due è un super-insieme definito dell'altra.

Linguaggio C++: peculiarità – I contenitori

Contenitore	Descrizione
	Sequenziali
vector	un array dinamico, simile all'array del C (per esempio, capace di accesso casuale) con la capacità di ridimensionarsi automaticamente a causa dell'inserimento o della cancellazione di elementi. Gli elementi sono memorizzati su una porzione di memoria continua. L'inserimento e la rimozione degli elementi nel/dal vector **in coda** viene effettuato in tempo costante (O(1)). L'inserimento e la rimozione all'inizio o nel centro e la ricerca vengono effettuate in tempo lineare (O(n)).
list	una lista bidirezionale; gli elementi non sono memorizzati in una memoria continua. Per questo motivo non è possibile accedere direttamente ad un elemento della lista accesso casuale, ma è necessario farlo tramite l'utilizzo di un iteratore. L'accesso agli elementi viene quindi effettuato con tempo lineare (O(n)) così come la ricerca, tuttavia le operazioni di inserimento e cancellazione vengono effettuate in tempo costante (O(1)).
	Associativi
set	un insieme ordinato che non consente duplicati; l'inserimento e la cancellazione degli elementi in un insieme non invalida il puntamento degli iteratori nell'insieme. Le operazioni sono l'unione, intersezione, differenza, differenza simmetrica e il test di inclusione.
multiset	come per il set, ma consente la presenza di elementi duplicati.
map	un array associativo ordinato rispetto alla chiave; consente la mappatura di un dato (chiave) associato ad un altro (valore). Entrambi i tipi di dato possono essere definiti dall'utente. Permette ricerche rapide rispetto alla chiave, l'accesso ai dati ha tempo logaritmico(O(log n)). Non consente di assegnare più chiavi ad un singolo valore.
multimap	come per la map, ma consente la presenza di chiavi duplicate.
hash_set **hash_multiset** **hash_map** **hash_multimap**	simili al set, multiset, map o multimap, rispettivamente, ma implementati usando una tabella hash; le chiavi non sono ordinate, ma una funzione hash deve esistere per ogni tipo di chiave. Questi contenitori non fanno parte della Libreria Standard C++, ma sono inclusi nella estensione SGI della STL, e sono comunemente incluse come per esempio nella libreria del GNU C++, nel namespace `__gnu_cxx` o nel namespace std_ext di Visual Studio. Potrebbero essere incluse nelle estensioni future dello standard C++.

Linguaggio C++: peculiarità – le classi **vector** e **list**

Quando abbiamo utilizzato il linguaggio C ed avevamo bisogno di di un **array** con **lunghezza variabile** dovevamo ricorrere all'**allocazione dinamica**, ricordandoci di allocare e deallocare opportunamente la memoria.

Stesso discorso valeva per le **liste linkate (o liste a puntatori)**, che dovevamo inoltre anche implementare da soli.

Fortunatamente il linguaggio **C++** ci viene incontro, offrendoci in automatico le implementazioni delle due classi speciali **vector** e **list**.

Un oggetto **vector** memorizza gli elementi che ne fanno parte in posizioni di memoria contigue.

Un oggetto **list** memorizza gli elementi che ne fanno parte in posizioni di memoria non contigue strutturando una lista doppiamente linkata.

Nel **linguaggio C++** per questo motivo **vector** e le **list** differiscono enormente nelle operazioni di **inserimento** e di **cancellazione**.

Entrambe le operazioni in una list sono molto più efficienti rispetto ad un vector perché per inserire un elemento all'inizio, alla fine o al centro si scambiano solo un paio di puntatori, mentre nel vettore occorre operare una serie di shift.

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

La classe **vector** ci offre array con dimensione variabile e relative funzioni di utility.

Sono molto comodi da utilizzare e non hanno particolari svantaggi rispetto agli array statici tradizionali.

I metodi di utility sono i seguenti:

begin() ritorna un **iteratore** che punta al primo elemento

end() ritorna un **iteratore** che punta all'ultimo elemento

size(): ritorna il **numero di elementi** presenti nel vettore

empty() ritorna **true** se il vettore è vuoto

push_back(T) **aggiunge** un elemento alla fine del vettore

pop_back() **rimuove** un elemento dall fine del vettore

insert(std::vector<T>::iterator, T) **aggiunge** l'elemento specificato nel secondo parametro alla posizione specificata dal primo

clear() elimina tutti gli elementi dal vettore

Linguaggio C++: peculiarità – la classe **vector - esempio**

10 BIS. vedi VECTOR\Pila.dev

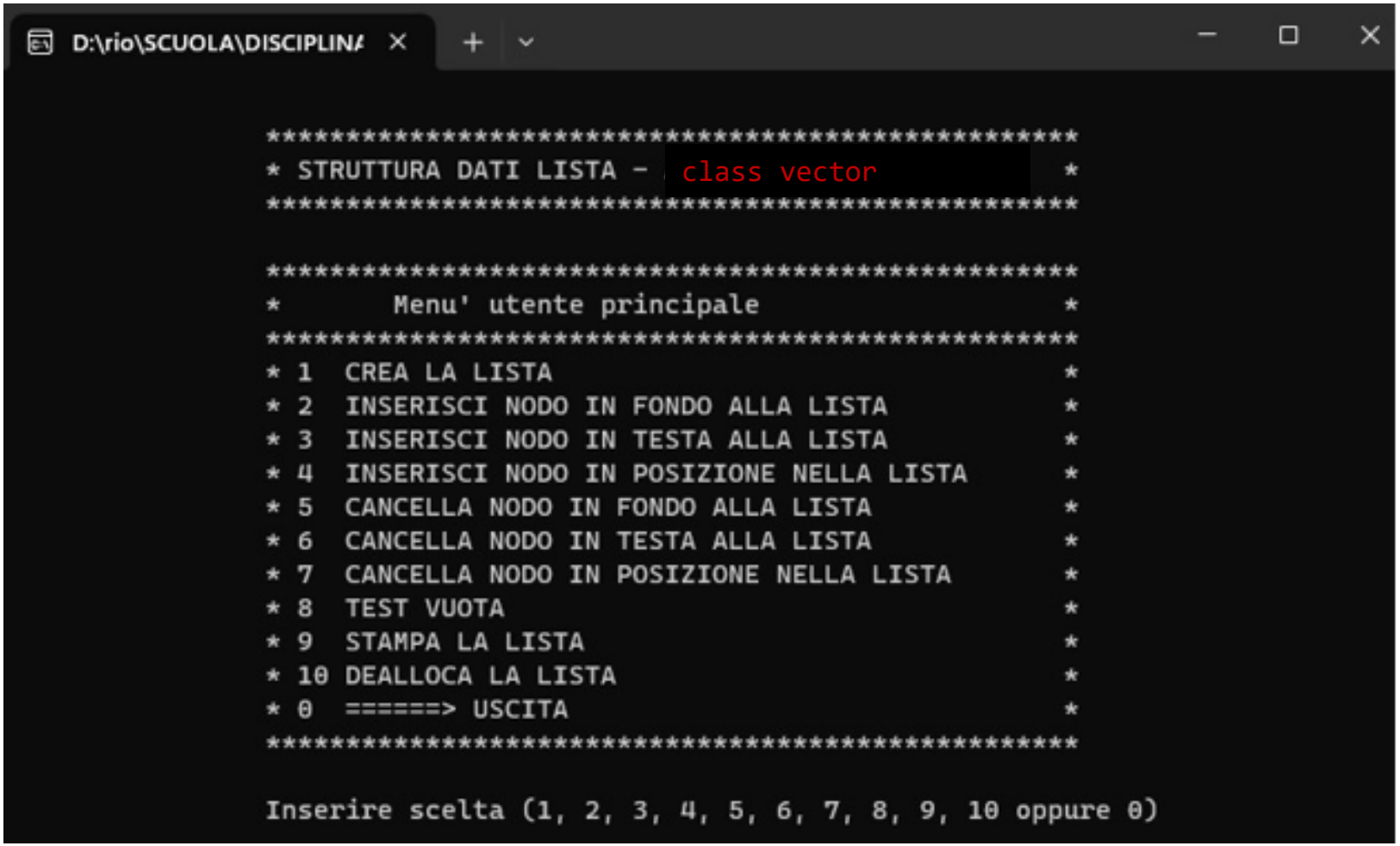
```
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, char** argv)
{
//Tra '<' e '>' metto il tipo del vector dinamico
vector<int> vettore;

//Aggiungo 3 numeri alla fine del vector
vettore.push_back(10);          // vettore = [10]
vettore.push_back(20);          // vettore = [10 20]
vettore.push_back(30);          // vettore = [10 20 30]

//Stampo il primo elemento del vettore
cout << vettore[0] << endl;    // vettore[0] = 10

//Elimino l'ultimo elemento dal vettore
vettore.pop_back();             // vettore = [10, 20]

//Inserisco 30 in seconda posizione
vettore.insert(vettore.begin() + 1, 30);  //vettore = [10 30 20]

//Stampo il contenuto del vettore
for (int i = 0; i < vettore.size(); i++)
   {
   cout << vettore[i] << " ";
   }
cout << endl;

//Svuoto il vettore
vettore.clear();

//Verifico se il vettore è vuoto
cout << "vuoto = " << vettore.empty();
return 0;
}
```

```
for (vector<int>::iterator j = vettore.begin(); j != vettore.end(); j++)
   {
   cout << *j << " ";
   }
cout << endl;
```

Un **iteratore** è un oggetto in grado di eseguire l'iterazione sugli elementi di un contenitore della libreria standard C++ e fornire accesso ai singoli elementi.
Tutti i **contenitori** della libreria template standard del C++ (come **vector** e **list**) forniscono iteratori che consentono agli algoritmi di accedere ai relativi elementi in maniera standard, senza doversi preoccupare del tipo di contenitore in cui sono archiviati gli elementi.

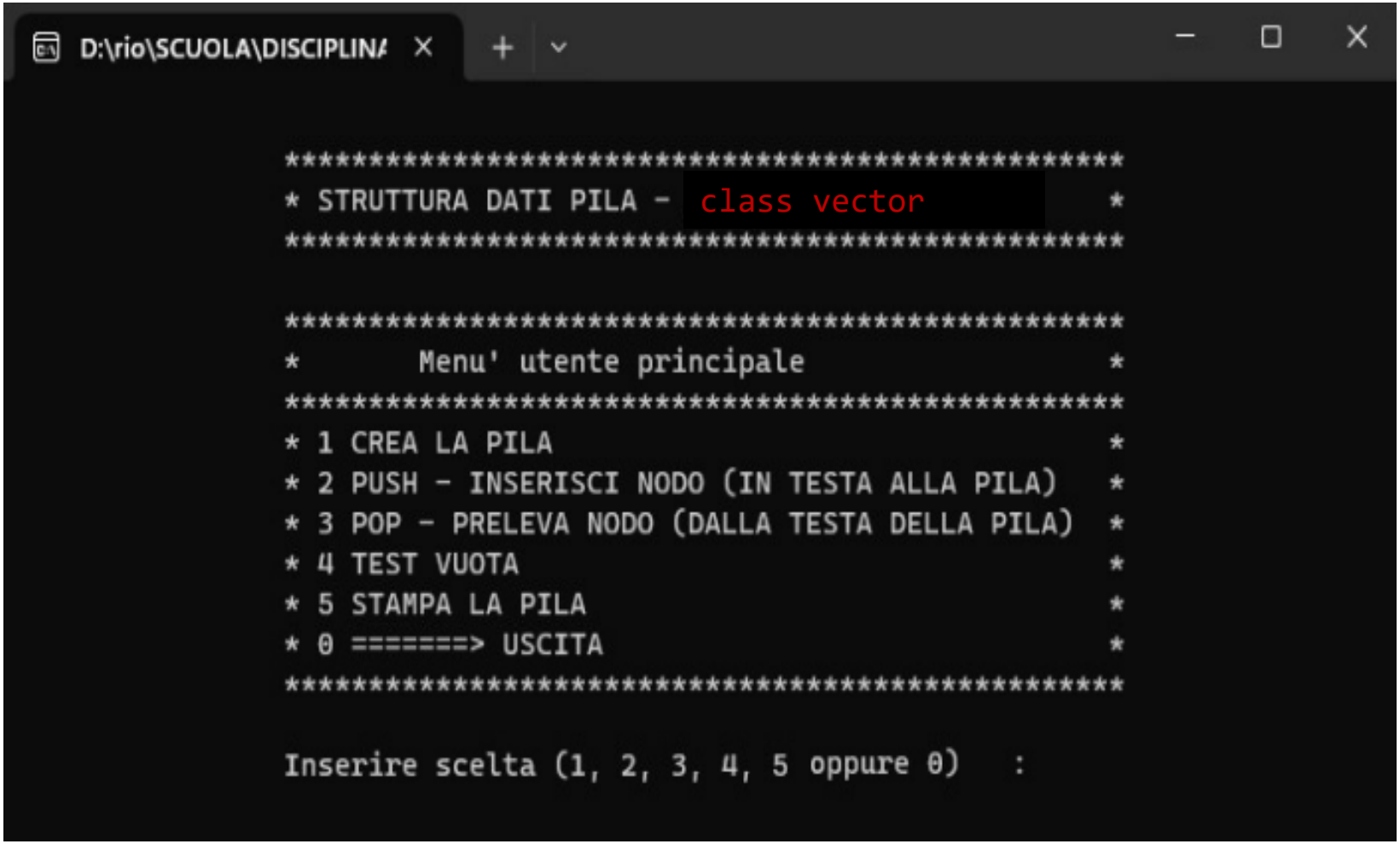
Linguaggio C++: peculiarità – Esercizi **EXTRA**

1) Utilizzando la classe **vector** implementa secondo l'ADT la struttura dati astratta **LISTA o SEQUENZA**



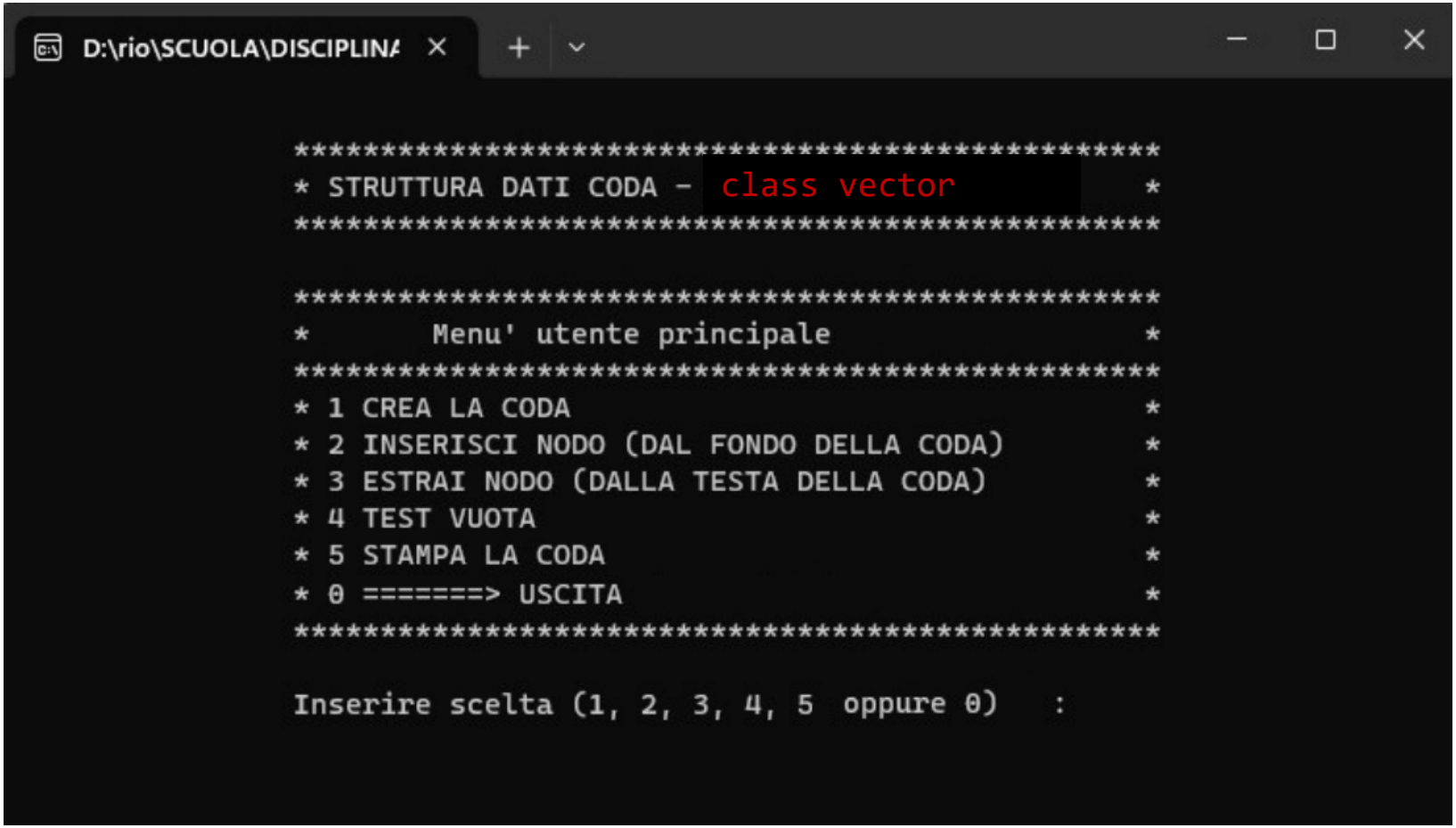
Linguaggio C++: peculiarità – Esercizi **EXTRA**

2) Utilizzando la classe **vector** implementa secondo l'ADT la struttura dati astratta **PILA o STACK**



Linguaggio C++: peculiarità – Esercizi **EXTRA**

3) Utilizzando la classe **vector** implementa secondo l'ADT la struttura dati astratta **CODA o QUEUE**



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

La classe **list** implementa una lista a doppio collegamento (doppiamentge linkata) con relative funzioni di utility.

Inserimento e cancellazione sono più performanti rispetto ai **vector**, ma manca l'accesso casuale, dato che l'allocazione in memoria degli elementi avviene su locazioni non contigue.

I metodi di utility sono i seguenti:

- **begin()** ritorna un **iteratore** che punta **al primo** elemento
- **end()** ritorna un **iteratore** che punta **all'ultimo** elemento
- **size()** ritorna il **numero di elementi** presenti nella lista
- **empty()** ritorna **true** se la lista è vuota
- **push_front(T)** **aggiunge** un elemento **all'inizio** della lista
- **push_back(T)** **aggiunge** un elemento **alla fine** della lista
- **pop_front()** **rimuove** un elemento **dall'inizio** della lista
- **pop_back()** **rimuove** un elemento **dalla fine** della lista
- **insert(std::list<T>::iterator, T)** **aggiunge** l'elemento specificato nel secondo parametro alla posizione specificata dal primo
- **clear()** elimina tutti gli elementi dalla lista

Linguaggio C++: peculiarità – la classe **list - esempio**

```cpp
#include <iostream>
#include <list>

using namespace std;

int main(int argc, char** argv)
{
//Tra '<' e '>' metto il tipo della lista
list<int> lista;
//Oggetto iteratore
list<int>::iterator i;

//Aggiungo 1 numero all'inizio della lista
lista.push_front(10);
//Aggiungo 2 numeri alla fine della lista
lista.push_back(20);
lista.push_back(30);

//Stampo il primo elemento della lista
cout << lista.front() << endl;
//Stampo l'ultimo elemento della lista
cout << lista.back() << endl;

//Elimino il primo elemento dalla lista
lista.pop_front();
//Elimino l'ultimo elemento dalla lista
lista.pop_back();
//Inserisco 30 all'inizio della lista
lista.insert(lista.begin(), 30);

//Stampo il contenuto della lista
for (i = lista.begin(); i != lista.end(); i++)
   {
    cout << *i << " ";
   }
cout << endl;

//Svuoto la lista
lista.clear();
//Verifico se la lista è vuota
cout << "vuoto = " << lista.empty();

return 0;
```

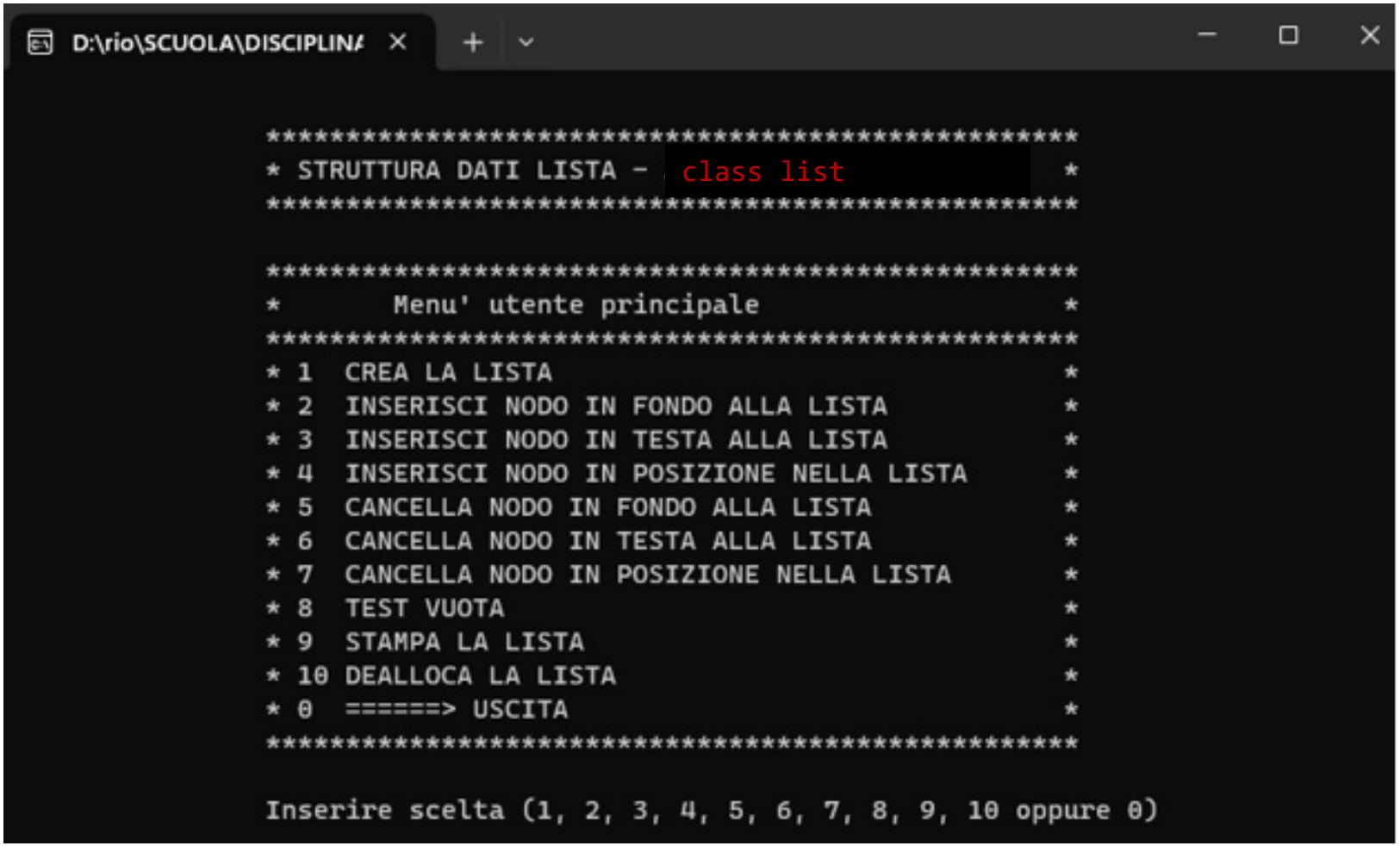
10 BIS. vedi LIST\Pila.dev

Un **iteratore** è un oggetto in grado di eseguire l'iterazione sugli elementi di un contenitore della libreria standard C++ e fornire accesso ai singoli elementi.
Tutti i **contenitori** della libreria template standard del C++ (come **vector** e **list**) forniscono iteratori che consentono agli algoritmi di accedere ai relativi elementi in maniera standard, senza doversi preoccupare del tipo di contenitore in cui sono archiviati gli elementi.

****** ATTENZIONE ! ******
Si fa notare che in questo caso l'uso di un **iteratore** è **OBBLIGATORIO** in quanto non è possibile utilizzare indici numerici per scorrere un oggetto della classe **list** (lista a puntatori linkata doppiamente).

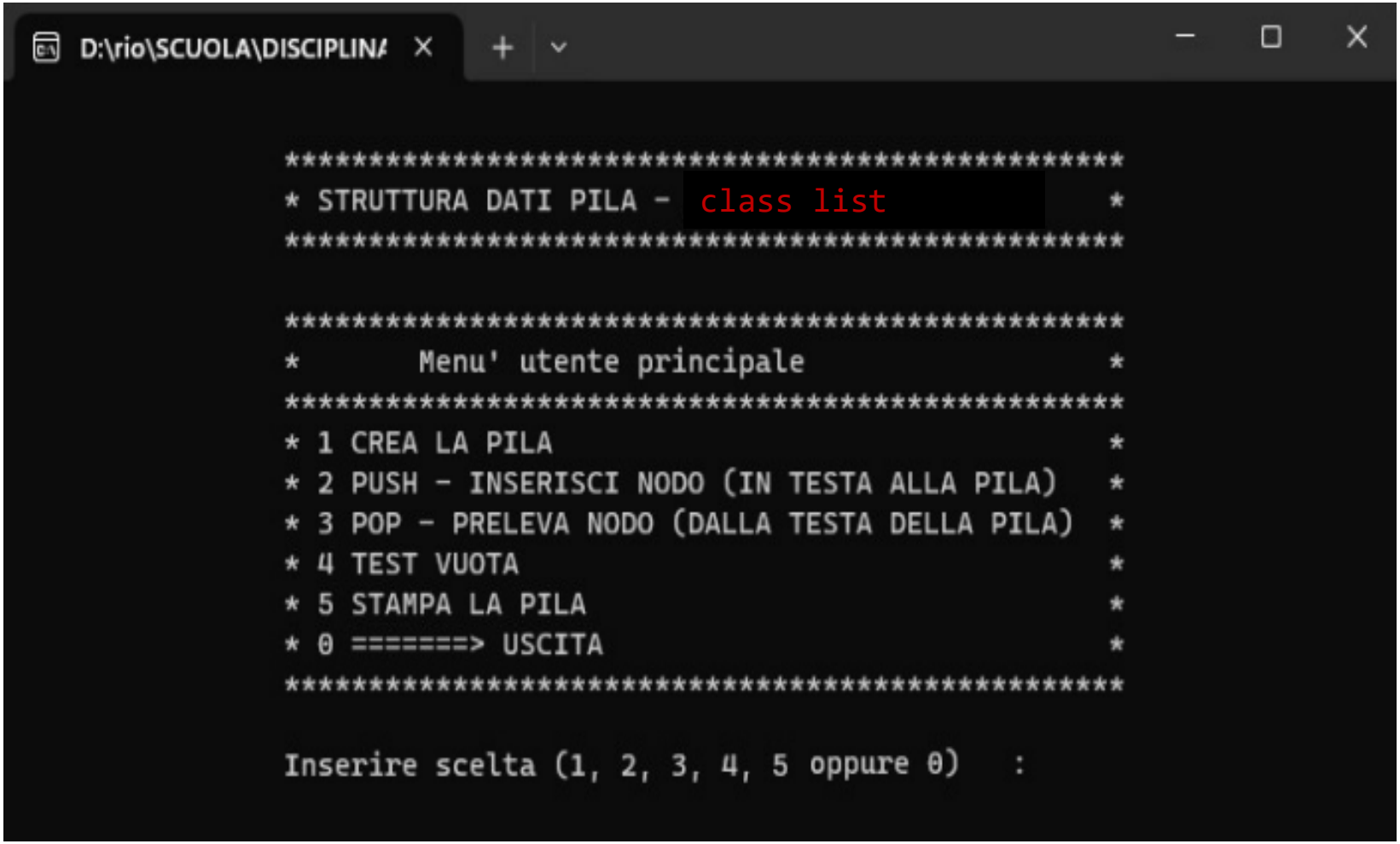
Linguaggio C++: peculiarità – Esercizi **EXTRA**

1) Utilizzando la classe **list** implementa secondo l'ADT la struttura dati astratta **LISTA o SEQUENZA**



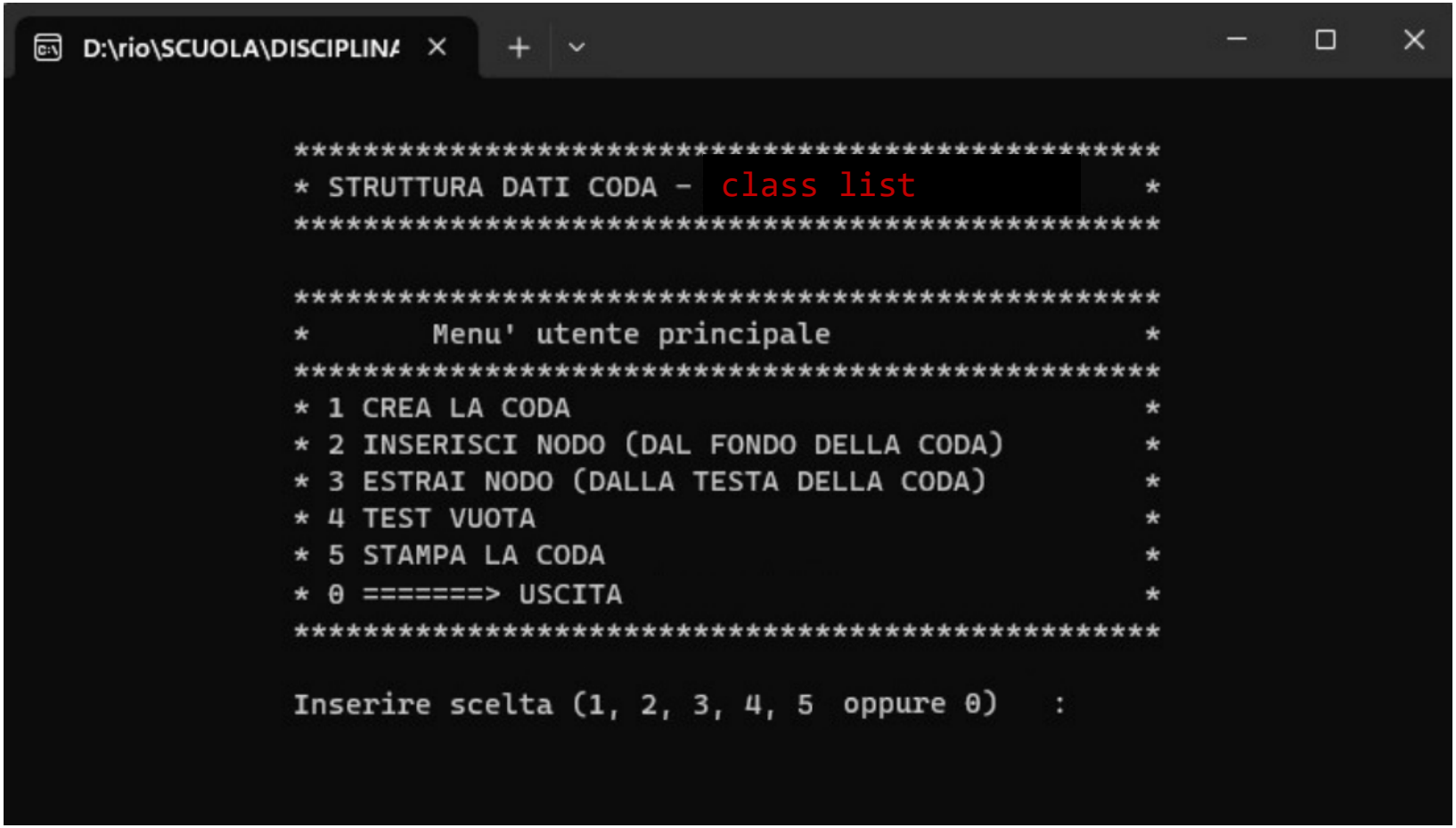
Linguaggio C++: peculiarità – Esercizi **EXTRA**

2) Utilizzando la classe **list** implementa secondo l'ADT la struttura dati astratta **PILA o STACK**



Linguaggio C++: peculiarità – Esercizi **EXTRA**

3) Utilizzando la classe **list** implementa secondo l'ADT la struttura dati astratta **CODA o QUEUE**



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 (**programmazione procedurale e/o modulare**).

Tale tipo di approccio si fonda sulla tecnica di programmazione **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 e/ modulare 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

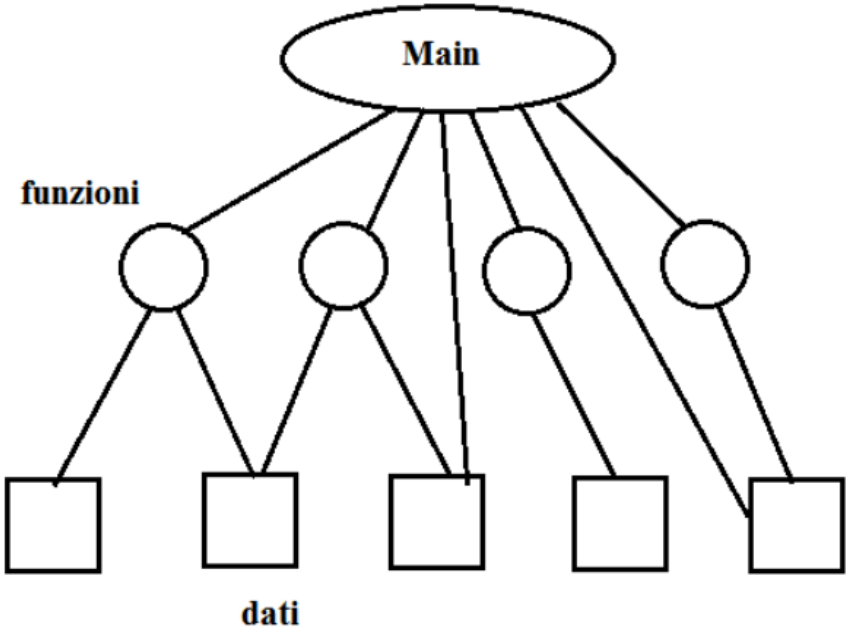
Uno degli **svantaggi principali** della programmazione procedurale e/o modulare è rappresentato dalla **manutenzione del programma**: spesso, per aggiungere o modificare parti di un programma **è necessaria la rielaborazione (**ricompilazione**) di tutto il programma stesso**

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

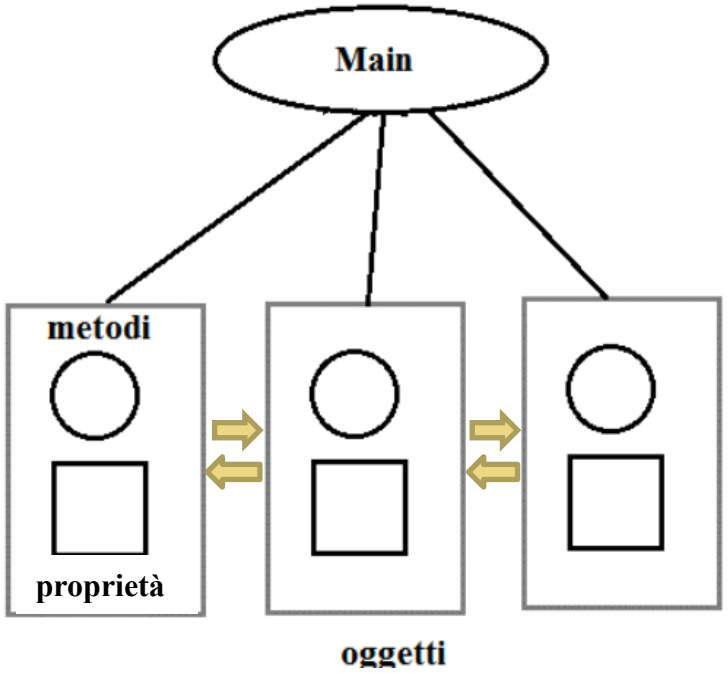
Linguaggio C++: OOP programming

- 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 (**proprietà** e **metodi** nel linguaggio C++)

Linguaggio C++: OOP programming



Programmazione procedurale e/o modulare



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 ad oggetti 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 metodi e/o proprietà alle classi degli oggetti coinvolti.

Se tale oggetti fanno parte di classi derivate erediteranno le proprietà delle classi 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)

Linguaggio C++: OOP programming principi base

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

Nel linguaggio C++ una "classe" rappresenta un tipo di dati astratto che può contenere elementi in stretta relazione tra loro che condividono le stesse "caratteristiche" (o proprietà) **e le stesse "azioni"** (o metodi)

Un **oggetto**, di conseguenza, è semplicemente una specifica istanza di una classe

In C++ 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 proprietà e metodi** 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 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.

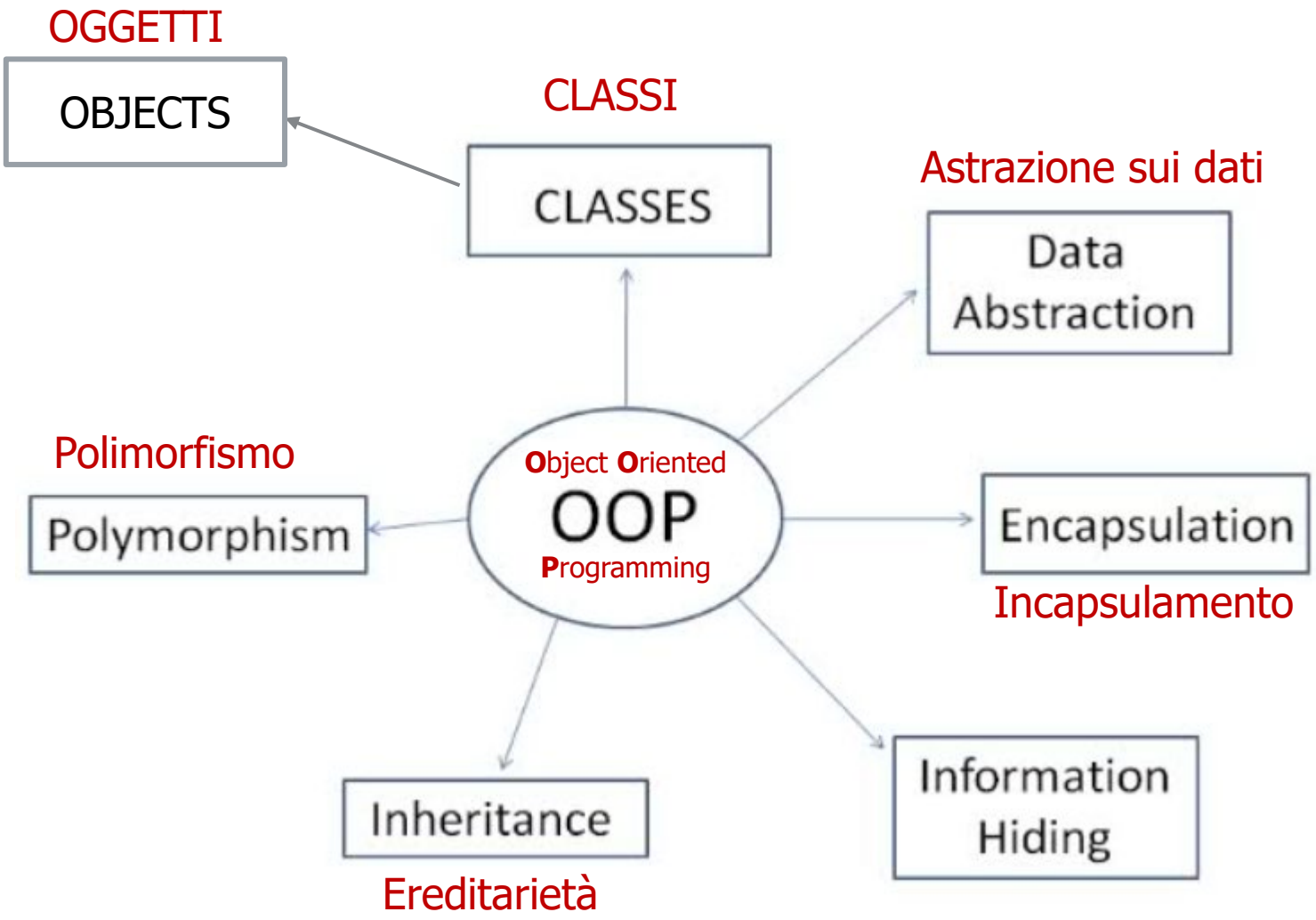
- Possibilità di overloading delle funzioni (si veda la definizione data nei capitoli precedenti).

- **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 (**metodi**) 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.

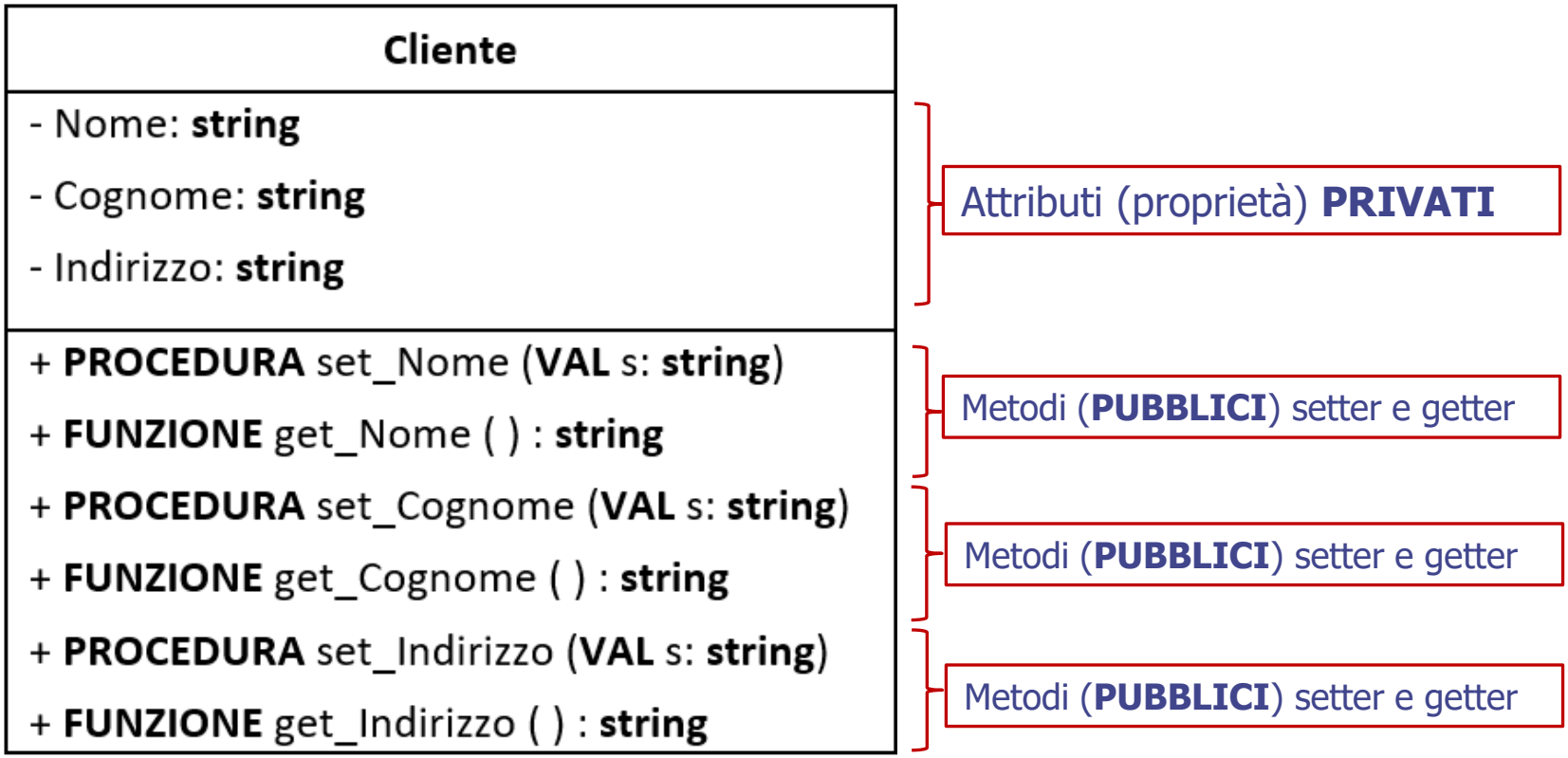
Linguaggio C++: OOP programming principi base



Progettazione di una classe

Progettiamo ora il primo esempio di classe **in C++:** classe **Cliente**

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



!!!ATTENZIONE!!!!

Costruire una classe con soli membri (proprietà e metodi) pubblici **non realizza appieno** uno dei principi basilari dell'OOP ossia **l'information hiding** ed è **una pratica da sconsigliare ASSOLUTAMENTE**

Progettazione dei **metodi** di una classe

```
PROCEDURA set_Nome (VAL s : string)
INIZIO
Nome ←s
RITORNA
FINE

FUNZIONE get_Nome () : string
INIZIO
RITORNA Nome
FINE

PROCEDURA set_Cognome (VAL s : string)
INIZIO
Cognome ←s
RITORNA
FINE
```

```
FUNZIONE get_Cognome () : string
INIZIO
RITORNA Cognome
FINE

PROCEDURA set_Indirizzo (VAL s : string)
INIZIO
Indirizzo ←s
RITORNA
FINE

FUNZIONE get_Indirizzo () : string
INIZIO
RITORNA Indirizzo
FINE
```

Linguaggio C++: implementazione di una classe

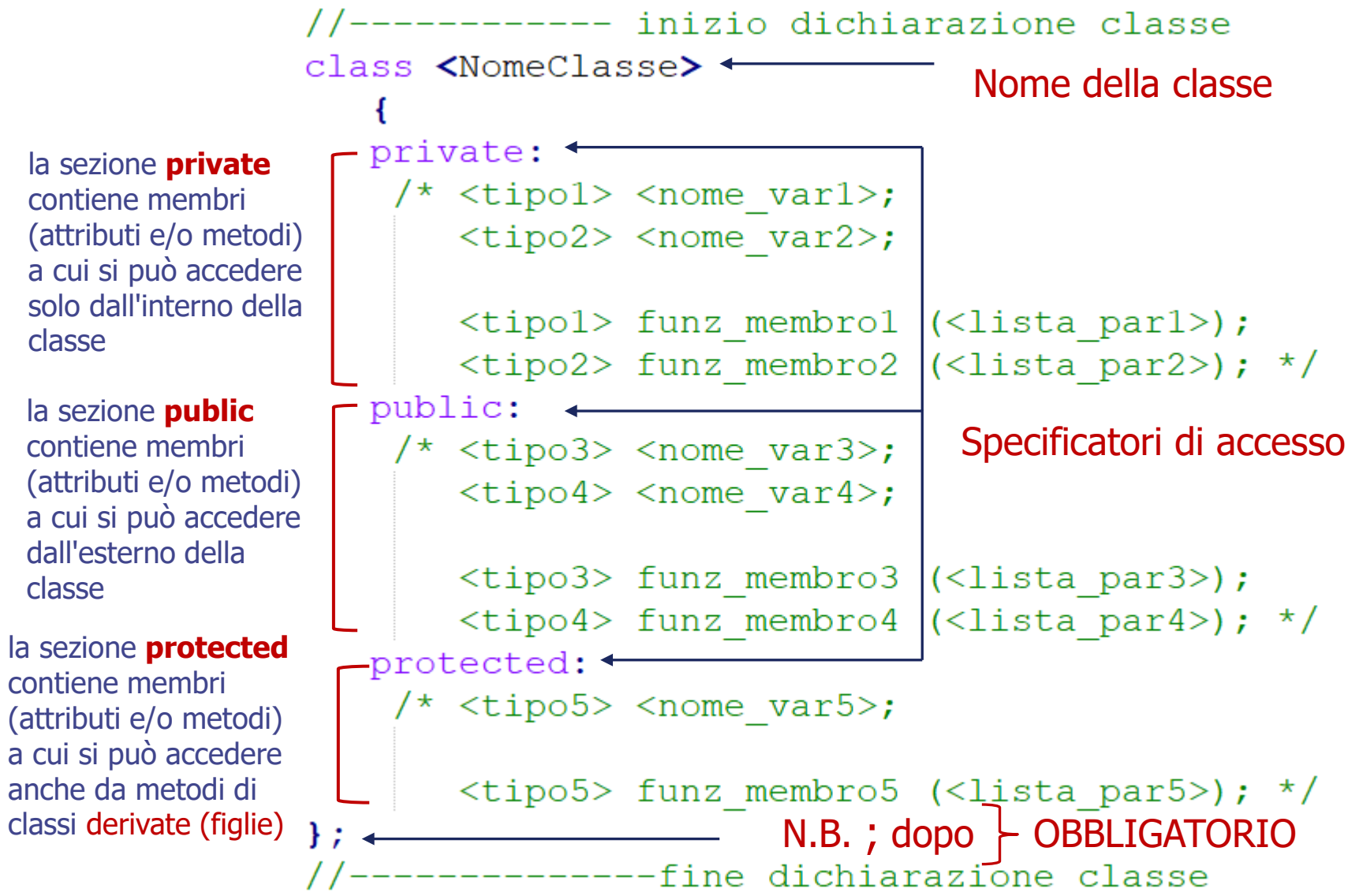
Per **definire una classe in C++** utilizziamo la parola riservata **class**

Subito dopo la parola riservata **class** deve essere indicato il **nome della classe** (ovvero il **nome del tipo di dato astratto**) e procediamo alla **definizione** della sua **struttura.**

La sua struttura va dettagliata nel blocco che segue la parola riservata **class** specificando:

- sia **l'interfaccia** della classe, ovvero le **proprietà** e i **metodi** (ovviamente **pubblici**) che gli oggetti metteranno a disposizione (esporranno) all'esterno;
- sia gli elementi **privati** e/o **protetti** che gli oggetti della classe riserveranno per se e/o per gli oggetti che da essi derivano

Vediamo ora un esempio in cui definiremo una classe e indicheremo con i commenti una possibile organizzazione dei suoi elementi……



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++: il puntatore this

La specifica del **linguaggio C++** include la parola riservata **this**, il cui uso è esclusivamente confinato all'ambito della definizione dei metodi di una classe.

Il tipo del puntatore **this** è quello di un puntatore all'oggetto corrente di una determinata classe.

Non occorre dichiararlo poiché la sua dichiarazione è **implicita nella classe** ossia del tipo:

nome_classe * **this**; //dove nome_classe è il tipo astratto identificato dalla classe

Il puntatore **this** contiene l'indirizzo dell'istanza della classe (oggetto) che ha in quel momento invocato un determinato metodo e può anche essere SOTTOINTESO ossia non utilizzato esplicitamente.

E' bene ricordare che il puntatore **this** è una propietà della classe a tutti gli effetti, che può anche essere **ereditata** e quindi esso viene reinizializzato nei costruttori delle classi derivate.

Linguaggio C++: primo programma con una classe

Per creare il nostro primo programma che utilizzi la classe **Cliente** sono possibili 3 diverse "strategie" implementative:

1) Sviluppo del progetto DEV-CPP su **un unico file cpp** (es. **ClienteAll.cpp**) contenente:

- la definizione della classe (metodi e proprietà);
- il **main** dove avverrà l'interazione degli oggetti della classe;
- l'implementazione dei suoi metodi.

2) Sviluppo del progetto DEV-CPP su **due file** contenenti rispettivamente:

- un header file (es. **Cliente.h**) contenente sia la definizione della classe (metodi e proprietà), sia l'implementazione dei suoi metodi;
- **il main** (es. **main.cpp**) dove avverrà sia l'implementazione dei suoi metodi, sia l'interazione degli oggetti.

3) Sviluppo del progetto DEV-CPP su **tre file** contenenti rispettivamente:

- **un header file** (es. **Cliente.h**) contenente la definizione della classe (metodi e proprietà);
- **un file cpp** (es. **Cliente.cpp**) contenente l'implementazione dei suoi metodi;
- **il main** (es. **main.cpp**) dove avverrà l'interazione degli oggetti.

In generale il progetto in caso di interazione di oggetti di più classi conterrà:

2 * numero delle classi + 1 (main) file

Linguaggio C++: primo programma con una classe

1) Sviluppo del progetto **DEV-CPP** su **un unico file cpp** (es. **ClienteAll.cpp**) contenente:

- la definizione della classe (metodi e proprietà);
-il **main** dove avverrà l'interazione degli oggetti;
- l'implementazione dei suoi metodi;

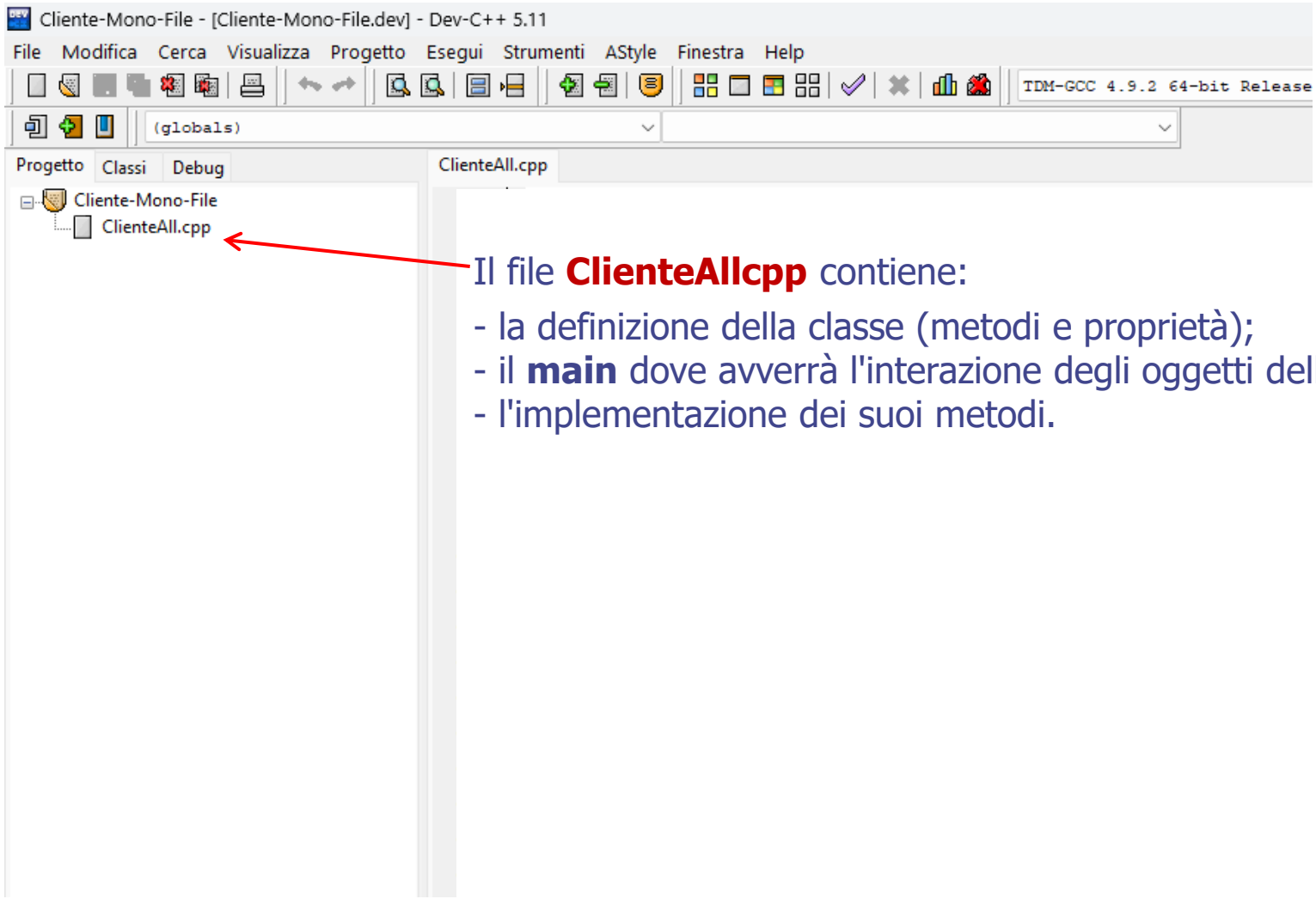
Questa modalità di scrittura della classe ha una valenza soprattutto didattica perché offre una veloce panoramica dell'intera struttura, ma **vincola** il codice della classe ad essere ospitato direttamente dal **main**.

L'enorme vantaggio deriva dal fatto che, dovendo gestire un unico file, inizialmente i programmi sembrano indiscutibimente più semplici da scrivere ma gli svantaggi sono molteplici (uno tra tutti l'impossibilità di poter affidare a sviluppatori diversi la codifica della classe e del main in quanto la risorsa la stessa).

11. vedi Cliente-Mono-File.dev

Linguaggio C++: primo programma con una classe

1) Sviluppo del progetto DEV-CPP **su un unico file: (Cliente-Mono-File.dev)**



Il file **ClienteAllcpp** contiene:

- la definizione della classe (metodi e proprietà);
- il **main** dove avverrà l'interazione degli oggetti della classe;
- l'implementazione dei suoi metodi.

Linguaggio C++: primo programma con una classe

1) Sviluppo del progetto DEV-CPP **su un unico file: (Cliente-Mono-File.dev)**

```
#include <iostream>
#include <string>

using namespace std;

//----- inizio dichiarazione classe Cliente
class Cliente
   {
   private:
      string Nome;
      string Cognome;
      string Indirizzo;
   public:
      void set_Nome(string s);
      string get_Nome( );

      void set_Cognome(string s);
      string get_Cognome( );

      void set_Indirizzo(string s);
      string get_Indirizzo( );
   };
//------ fine dichiarazione classe Cliente
```

Il file **ClienteAll.cpp** contiene:

- la definizione della classe (metodi e proprietà);

.....

.....

Linguaggio C++: primo programma con una classe

1) Sviluppo del progetto DEV-CPP **su un unico file: (Cliente-Mono-File.dev)**

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

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

cout << "Nome del cliente e': " << c.get_Nome() << endl;
cout << "Cognome del cliente e': " << c.get_Cognome() << endl;
cout << "Indirizzo del cliente e': " << c.get_Indirizzo() << endl;

return 0;
}
//------ fine main
```

Definizione di un oggetto della classe **Cliente** (Allocazione STATICA --- Segmento STACK)

Accesso ai suoi metodi/proprietà tramite il nome dell'oggetto attraverso l'operatore **punto .**

Il file **ClienteAll.cpp** contiene:

……

- il **main** dove avverrà l'interazione degli oggetti della classe;

……

Oggetto di una classe allocato DINAMICAMENTE

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++: primo programma con una classe

1) Sviluppo del progetto DEV-CPP **su un unico file: (Cliente-Mono-File.dev)**

```
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** (Allocazione DINAMICA --- Segmento HEAP)

Accesso ai suoi metodi/proprietà tramite puntatore ad un oggetto attraverso l'operatore freccia **->**

Deallocazione dinamica di un oggetto della classe **Cliente**

Linguaggio C++: primo programma con una classe

1) Sviluppo del progetto DEV-CPP **su un unico file: (Cliente-Mono-File.dev)**

```
//----- inizio metodi PUBBLICI classe Cliente
void Cliente::set_Nome(string s)
{
this->Nome = s;
return;
}

string Cliente::get_Nome( )
{
return this->Nome;
}

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

string Cliente::get_Cognome( )
{
return this->Cognome;
}

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

string Cliente::get_Indirizzo( )
{
return this->Indirizzo;
}
//----- fine metodi PUBBLICI classe Cliente
```

tipo_restituito nome_classe::nome_metodo (eventuali parametri)

Il file **ClienteAll.cpp** contiene:

.....

.....

- l'implementazione dei metodi della classe **Cliente**

!!!ATTENZIONE!!!!

Il puntatore implicito **this** all'oggetto corrente della classe potrebbe essere omesso in quanto sottointeso ed il codice "funzionerebbe" lo stesso!

Linguaggio C++: primo programma con una classe

2) Sviluppo del progetto **DEV-CPP** su **due file** contenenti rispettivamente:

- un header file (es. **Cliente.h**) contenente sia la definizione della classe (metodi e proprietà) sia l'implementazione dei suoi metodi;
- **il main** (es. **main.cpp**) dove avverrà l'interazione degli oggetti.

Questa modalità di scrittura della classe, come quella vista in precedenza ha ancora una valenza soprattutto didattica perché offre una veloce panoramica dell'intera struttura, anche se **svincola** il codice dei metodi della classe ad essere implementati direttamente nel **main**.

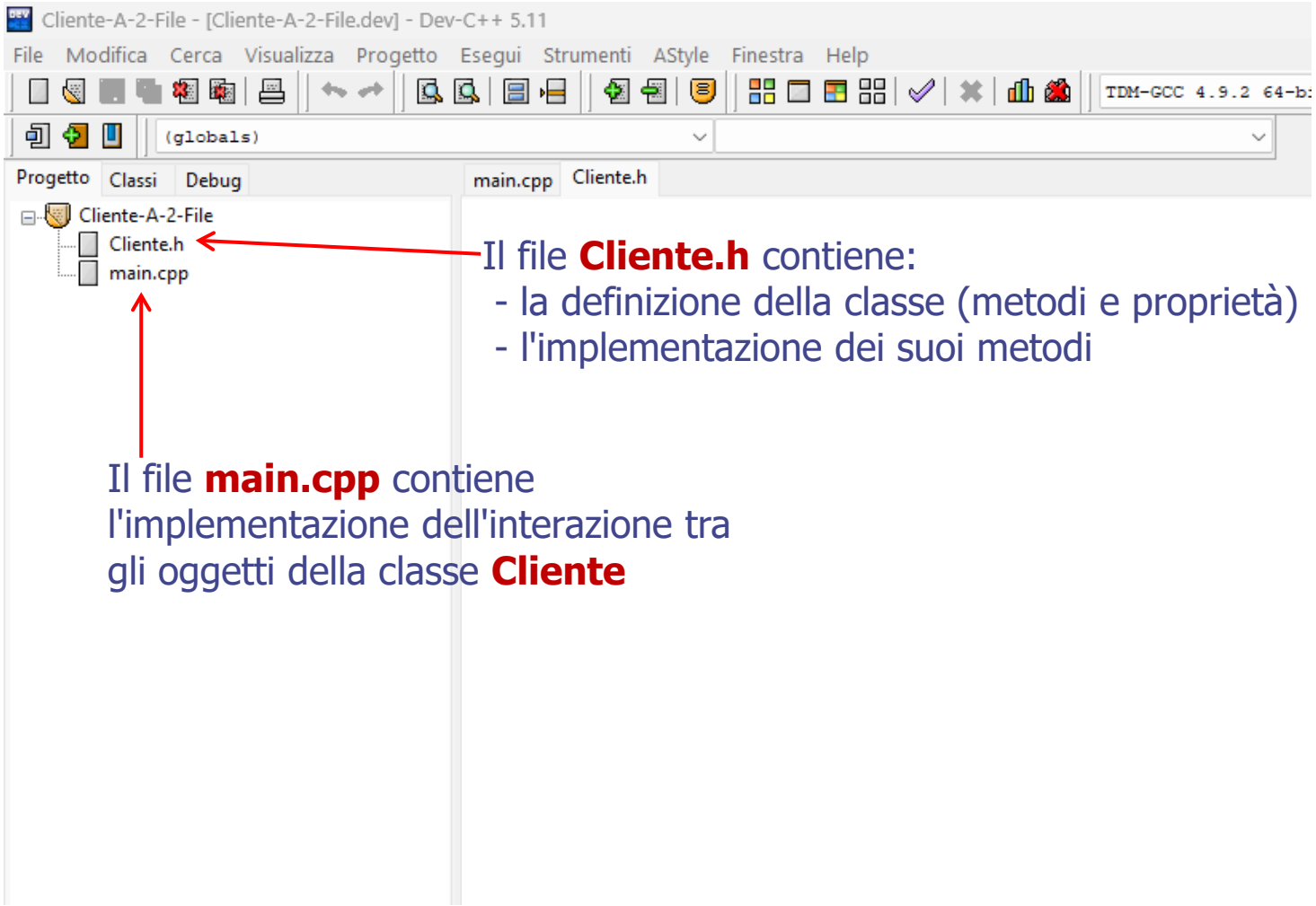
In questo caso anche se si gestiscono solo due file ed i programmi sembrano ancora semplici da scrivere (rendendo possibile poter suddividere gli sviluppatori tra **classe** e **main** in quanto risorse differenti), il **file header** della classe **Cliente.h** potrebbe tendere ad essere abbastanza complesso in presenza di un numero elevato di metodi costituiti magari da molte linee di codice.

Questo tenderebbe a rendere la struttura complesiva della classe di difficile comprensione.

11. vedi Cliente-A-2-File.dev

Linguaggio C++: primo programma con una classe

2) Sviluppo del progetto DEV-CPP su **due file**: **(Cliente-A-2-File.dev)**



Linguaggio C++: primo programma con una classe

2) Sviluppo del progetto DEV-CPP su **due file**: **(Cliente-A-2-File.dev)**

tipo_restituito nome_classe::nome_metodo (eventuali parametri)

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

//----- inizio dichiarazione classe Cliente
class Cliente
   {
   private:
      string Nome;
      string Cognome;
      string Indirizzo;

   public:
      //----- inizio metodi PUBBLICI classe
      void set_Nome(string s)
      {
      Nome = s;
      return;
      }
      string get_Nome( )
      {
      return Nome;
      }
      void set_Cognome(string s)
      {
      Cognome = s;
      return;
      }
      string get_Cognome( )
      {
      return Cognome;
      }
      void set_Indirizzo(string s)
      {
      Indirizzo = s;
      return;
      }
      string get_Indirizzo()
      {
      return Indirizzo;
      }
      //----- fine metodi PUBBLICI classe Cliente
   };
//------ fine dichiarazione classe Cliente
```

Il file **header Cliente.h** contiene:
- la definizione della classe (metodi e proprietà)
- l'implementazione dei suoi metodi

Linguaggio C++: primo programma con una classe

2) Sviluppo del progetto DEV-CPP su **due file**: **(Cliente-A-2-File.dev)**

```
#include <iostream>
using namespace std;
#include "Cliente.h"
//------ inizio main
int main(int argc, char** argv)
{
Cliente c;
string s1, s2, s3;

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

cout << "Nome del cliente e': " << c.get_Nome() << endl;
cout << "Cognome del cliente e': " << c.get_Cognome() << endl;
cout << "Indirizzo del cliente e': " << c.get_Indirizzo() << endl;

return 0;
}
//------ fine main
```

Definizione di un oggetto della classe **Cliente** (Allocazione STATICA --- Segmento STACK)

Accesso ai suoi metodi/proprietà tramite il nome dell'oggetto attraverso l'operatore **punto .**

Il file **main.cpp** contiene:
- il **main** dove avverrà l'interazione degli oggetti della classe;

Linguaggio C++: primo programma con una classe

2) Sviluppo del progetto DEV-CPP su **due file**: **(Cliente-A-2-File.dev)**

```
#include <iostream>
using namespace std;
#include "Cliente.h"
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** (Allocazione DINAMICA --- Segmento HEAP)

Accesso ai suoi metodi/proprietà tramite puntatore ad un oggetto attraverso l'operatore freccia **->**

Deallocazione dinamica di un oggetto della classe **Cliente**

Linguaggio C++: primo programma con una classe

3) Sviluppo del progetto **DEV-CPP** su **tre file** contenenti rispettivamente:

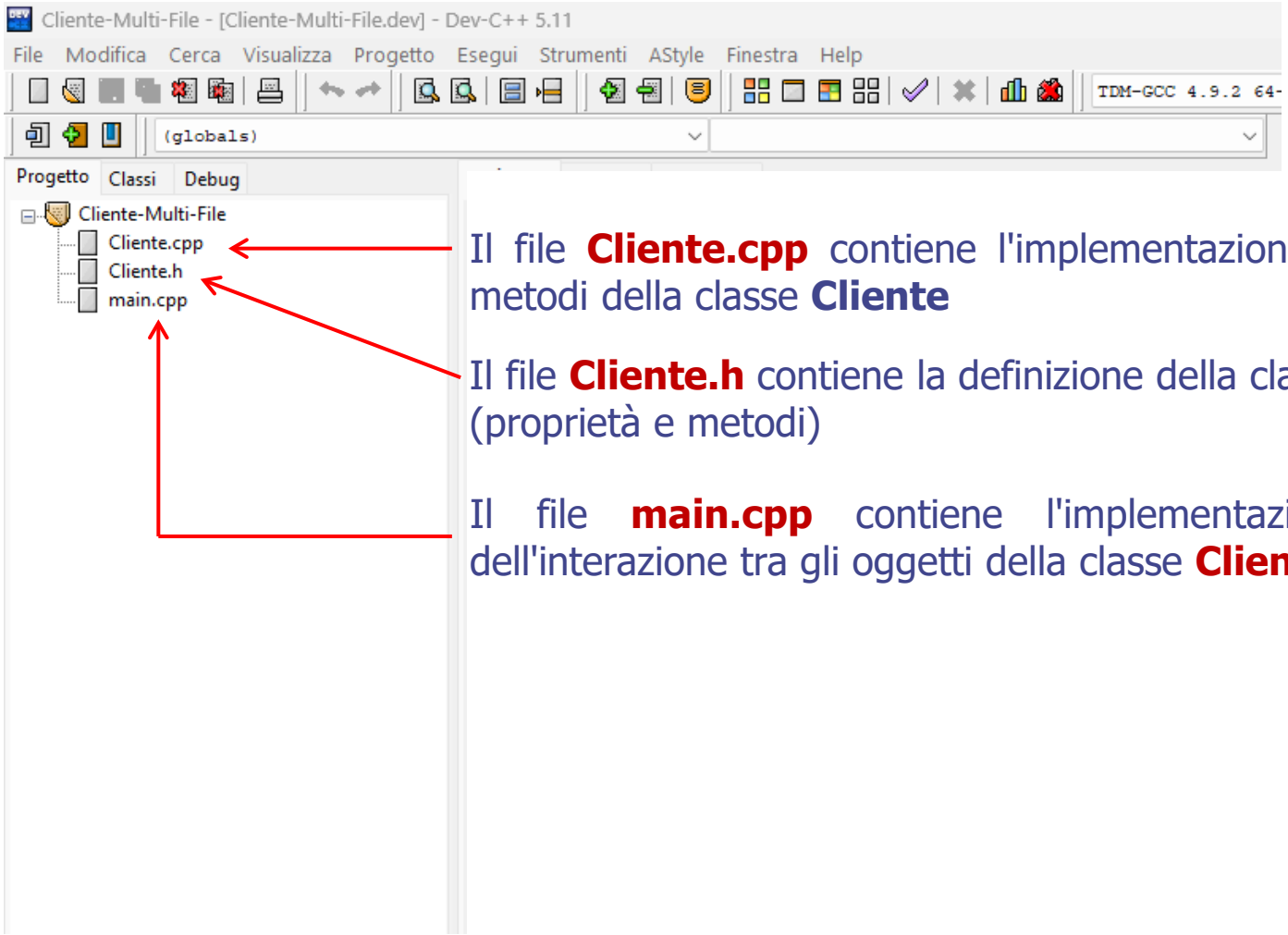
-**un header file** (es. **Cliente.h**) contenente la definizione della classe (metodi e proprietà);
- **un file cpp** (es. **Cliente.cpp**) contenente l'implementazione dei suoi metodi;
-**il main** (es. **main.cpp**) dove avverrà l'interazione degli oggetti.

Questa modalità, OVVIAMENTE, è la più indicata per progetti articolati con molte classi da utilizzare rendendo possibile agli sviluppatori coinvolti il lavorare **modularmente** e pienamente in **parallelo**.

11. vedi Cliente-Multi-File.dev

Linguaggio C++: primo programma con una classe

3) Sviluppo del progetto DEV-CPP **su tre file**: **(Cliente-Multi-File.dev)**:



Il file **Cliente.cpp** contiene l'implementazione di tutti i metodi della classe **Cliente**

Il file **Cliente.h** contiene la definizione della classe **Cliente** (proprietà e metodi)

Il file **main.cpp** contiene l'implementazione dell'interazione tra gli oggetti della classe **Cliente**

Linguaggio C++: definizione di una classe

In quest'ultimo caso il file **Cliente.h** potrebbe essere fatto così:

```
#ifndef CLIENTE_H
#define CLIENTE_H

#include <iostream>
using namespace std;

//----- inizio dichiarazione classe Cliente
class Cliente
   {
   private:
      string Nome;
      string Cognome;
      string Indirizzo;
   public:
      void set_Nome(string s);
      string get_Nome( );

      void set_Cognome(string s);
      string get_Cognome( );

      void set_Indirizzo(string s);
      string get_Indirizzo( );
   };
//------ fine dichiarazione classe Cliente

#endif
```

Nota. La direttiva **#ifndef** è utile per verificare se un simbolo è già stato dichiarato ed evitare di dichiararlo più volte nello stesso codice sorgente. Quando un progetto è articolato su più file che utuilizzando magari più classi può diventare problematico evitare la multi-inclusione

N.B. Sono privati anche se non specificato **private**:

IN OGNI CASO E' MEGLIO DICHIARARE ESPLICITAMENTE LE PROPRIETA' ED I METODI PRIVATI UTILIZZANDO L'APPOSITO SPECIFICATORE **private**

Nota. La direttiva **#ifndef** va chiusa OBBLIGATORAMENTE con **#endif**

Linguaggio C++: definizione dei metodi di una classe

```
#include "Cliente.h"
//------ inizio metodi PUBBLICI classe Client
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;
}
//------ fine metodi PUBBLICI classe Cliente
```

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

Si noti la particolare sintassi utilizzata nel file **Cliente.cpp** relativamente alla implementazione delle funzioni membro (metodi). Essa segue la regola:

tipo_restituito nome_classe::nome_metodo (eventuali parametri)

in quanto al nome di ogni metodo della classe **Cliente** si deve anteporre la scrittura

Cliente::

poichè occorre specificare l'ambito di visibilità (**scope resolution**) delle sue funzioni membro (metodi)

Linguaggio C++: definizione del main

PRIMA IPOTESI: Supponiamo di volere utilizzare un oggetto della classe Cliente allocato staticamente

Il file **main.cpp** potrebbe essere fatto così

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

#include "Cliente.h"

//------ inizio main
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 << "Nome del cliente e': " << c.get_Nome() << endl;
cout << "Cognome del cliente e': " << c.get_Cognome() << endl;
cout << "Indirizzo del cliente e': " << c.get_Indirizzo() << endl;

return 0;
}
//------ fine main
```

Definizione di un oggetto della classe **Cliente** (Allocazione STATICA --- Segmento STACK)

Accesso ai suoi metodi/proprietà tramite il nome dell'oggetto attraverso l'operatore **punto .**

Linguaggio C++: definizione di una classe

SECONDA IPOTESI: Supponiamo di volere utilizzare un oggetto della classe Cliente allocato dinamicamente

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

#include "Cliente.h"

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** (Allocazione DINAMICA --- Segmento HEAP)

Accesso ai suoi metodi/proprietà tramite puntatore ad un oggetto attraverso l'operatore freccia **->**

Deallocazione dinamica di un oggetto della classe **Cliente**

Linguaggio C++: definizione di una classe

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

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

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). questo sarebbe chiaramente un errore anche se il compilatore **NON** ce lo farebbe notare.

Linguaggio C++: Il **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).

Una classe può avere, se necessario, più COSTRUTTORI!

Infatti si parla in questo caso di OVERLOAD dei costruttori

Linguaggio C++: Il **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).

Una classe HA SEMPRE UN UNICO DISTRUTTORE!

Classe **Cliente**: definizione con **Costruttore** e **Distruttore**

Esercizio: Proviamo ad implementare la **versione definitiva** della classe **Cliente** provando a dichiarare ed a valorizzare più oggetti sia staticamente sia dinamicamente utilizzando anche **l'overloading** dei 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** ( ) : **Cliente** + **COSTRUTTORE** **Cliente** (**VAL** s1: **string** , **VAL** s2: **string** , **VAL** s3: **string**) : **Cliente** + **DISTRUTTORE** **~Cliente** ( )

12. vedi Cliente-Multi-File-Overloading-Costruttore.dev

Autore: Prof. Rio Chierego

Classe **Cliente**: i **metodi** della classe

```
PROCEDURA set_Nome (VAL s : string)
INIZIO
Nome ←s
RITORNA
FINE

FUNZIONE get_Nome () : string
INIZIO
RITORNA Nome
FINE

PROCEDURA set_Cognome (VAL s : string)
INIZIO
Cognome ←s
RITORNA
FINE
```

```
FUNZIONE get_Cognome () : string
INIZIO
RITORNA Cognome
FINE

PROCEDURA set_Indirizzo (VAL s : string)
INIZIO
Indirizzo ←s
RITORNA
FINE

FUNZIONE get_Indirizzo () : string
INIZIO
RITORNA Indirizzo
FINE
```

Classe **Cliente**: i **Costruttori** ed il **Distruttore**

COSTRUTTORE Cliente ( ) : **Cliente**
oggetto : **Cliente**
INIZIO
Scrivi ("Costruttore 1 (default ritoccato) - allocato oggetto della classe Cliente")
oggetto.Nome ← "MARIO"
oggetto.Cognome ← "ROSSI"
oggetto.Indirizzo ← "VIA VAI n. 180"
RITORNA **oggetto**
FINE

COSTRUTTORE Cliente (**VAL** string s1, **VAL** string s2, **VAL** string s3) : **Cliente**
oggetto : **Cliente**
INIZIO
Scrivi ("Costruttore 2 (overloading) - allocato oggetto della classe Cliente)
oggetto.Nome ← s1
oggetto.Cognome ← s2
oggetto.Indirizzo ← s3
RITORNA **oggetto**
FINE

N.B. IN FASE DI PROGETTAZIONE la parola chiave "**COSTRUTTORE**" è da intendersi come **ALIAS** di "**FUNZIONE**" ritornando **SEMPRE** un oggetto della classe cui si riferisce mentre la parola chiave **DISTRUTTORE** è da intendersi come **ALIAS** di "**PROCEDURA**" non potendo **MAI** ritornare esplicitamente alcunchè nel proprio nome!
IN FASE DI PROGRAMMAZIONE sia i **COSTRUTTORI** sia il **DISTRUTTORE** saranno funzioni che non restituiscono niente (void quindi **PROCEDURE**)

DISTRUTTORE ~Cliente ( )
INIZIO
Scrivi ("Distruttore (UNICO) - deallocato oggetto della classe Cliente")
RITORNA
FINE

Classe **Cliente**: Linguaggio C++ - file **Cliente.h**

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

//----- inizio dichiarazione classe Cliente
class Cliente
{
private:
    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();

    Cliente();                  // 1° costruttore della classe (DEFAULT ritoccato)
    Cliente(string, string, string);   // 2° costruttore della classe (OVERLOADING)
    ~Cliente();
};
//----- fine dichiarazione classe Cliente
```

Classe **Cliente**: Linguaggio C++ - file **Cliente.cpp**

```
#include "Cliente.h"

//----- inizio metodi PUBBLICI classe Cliente
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;
}
```

Classe **Cliente**: Linguaggio C++ - file **Cliente.cpp**

```
// 1° Costruttore DEFAULT
Cliente::Cliente()
{
cout << "Costruttore 1 (default ritoccato) - allocato oggetto della casse Cliente" << endl;
Nome = "MARIO";
Cognome = "ROSSI";
Indirizzo = "VIA VAI n. 180";
return;
}

// 2° Costruttore
Cliente::Cliente(string s1, string s2, string s3)
{
cout << "Costruttore 2 (overloading) - allocato oggetto della classe Cliente" << endl;

Nome = s1;
Cognome = s2;
Indirizzo = s3;
return;
}

//Distruttore
Cliente::~Cliente()
{
cout << endl << "Distruttore (UNICO) - deallocato oggetto della classe Cliente" << endl;
return;
}
//----- fine metodi PUBBLICI classe Cliente
```

Classe **Cliente**: Linguaggio C++ - file **main.cpp**

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

#include "Cliente.h"

//----- inizio main
int main(int argc, char** argv)
{
Cliente c;                  // 1° modo per allocare un Oggetto della classe Cliente (stack)
                            // viene chiamato il costruttore di default ritoccato (SENZA PARAMETRI)
//Cliente c = Cliente();    // modo alternativo per allocare un Oggetto della classe Cliente (stack)
                            // viene chiamato il costruttore di default ritoccato (SENZA PARAMETRI)

Cliente c1 = Cliente ("Giacomo", "Puccini", "Via Verdi n. 115"); //2°  modo - COSTRUTTORE CON PARAMETRI

string s1, s2, s3;

//Visualizzazione oggetto cliente c (dati impostati di default)
cout << "******** Dati inseriti Oggetto cliente c (STATICO) 1^ costruttore **********" << endl;
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;
```

Classe **Cliente**: Linguaggio C++ - file **main.cpp**

```
//Gestione modifica dati oggetto cliente c
cout << endl << "********** Dati per modificare lo stato dell''oggetto cliente c (STATICO) ***********" << endl;
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 << "******** Dati modificati Oggetto cliente c (STATICO) 1^ costruttore ***********" << endl;
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;

//Gestione oggetto cliente c1
cout << "******** Dati inseriti Oggetto cliente c1 (STATICO) 2^ costruttore ***********" << endl;
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 << endl;

//Gestione oggetto cliente c2
Cliente c2 = Cliente (s1, s2, s3);  //2°  modo - COSTRUTTORE CON PARAMETRI

//Gestione oggetto cliente c2
cout << "*** Dati inseriti Oggetto cliente c2 (STATICO) 2^ costruttore ***" << endl;
cout << "Il nome del cliente inserito e': " << c2.get_Nome() << endl;
cout << "Il cognome del cliente inserito e': " << c2.get_Cognome() << endl;
cout << "L' indirizzo del cliente inserito e': " << c2.get_Indirizzo() << endl << endl;
```

Classe **Cliente**: Linguaggio C++ - file **main.cpp**

```
//Gestione oggetto cliente c3 - allocazione dinamica
Cliente* c3 = new Cliente();

//Visualizzazione oggetto cliente c3 (dati impostati di default)
cout << "******** Dati inseriti Oggetto cliente c (STATICO) 1^ costruttore **********" << endl;
cout << "Il nome del cliente inserito e': " << c3->get_Nome() << endl;
cout << "Il cognome del cliente inserito e': " << c3->get_Cognome() << endl;
cout << "L' indirizzo del cliente inserito e': " << c3->get_Indirizzo() << endl;

cout << endl << "*** Dati per  modificare lo stato dell'oggetto cliente c3 (DINAMICO) ***" << endl;
cout << "Nome : ";
cin >> s1;
c3->set_Nome(s1);
cout << "Cognome : ";
cin >> s2;
c3->set_Cognome(s2);
cout << "Indirizzo : ";
getline(cin,s3);
c3->set_Indirizzo(s3);

cout << "***** Dati modificati Oggetto cliente c3 (DINAMICO) 1^ costruttore ****" << endl;
cout << "Il nome del cliente inserito e': " << c3->get_Nome() << endl;
cout << "Il cognome del cliente inserito e': " << c3->get_Cognome() << endl;
cout << "L' indirizzo del cliente inserito e': " << c3->get_Indirizzo()<< endl;

//deallocazione oggetto c3
delete(c3);

//Gestione oggetto cliente c4 -  allocazione dinamica
// Cliente* c4 = ew Cliente(s1, s2, s3);
Cliente* c4 = new Cliente("Marie", "Curie", "Via Po, 8");

cout << "***** Dati inseriti Oggetto cliente c4 (DINAMICO) 2^ costruttore ****" << endl;
cout << "Il nome del cliente inserito e': " << c4->get_Nome() << endl;
cout << "Il cognome del cliente inserito e': " << c4->get_Cognome() << endl;
cout << "L' indirizzo del cliente inserito e': " << c4->get_Indirizzo()<< endl;

//deallocazione oggetto c4
delete(c4);
return 0;
}
//----- fine main//----- fine main
```

Classe **Cliente**: **Estensione proprietà e metodi**

1) Aggiungi alla classe Cliente la proprietà Sesso (privata) definita come singolo carattere da valorizzare con:

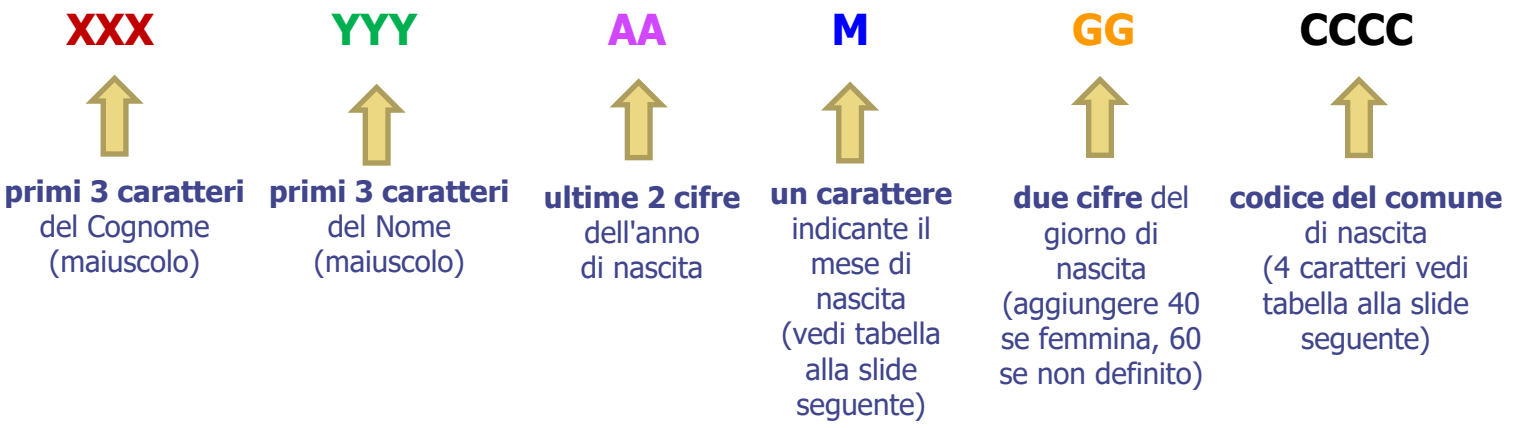
- **'M' per maschio;**
- **'F' per femmina;**
- **'X' per non definito;**

nonché i relativi metodi *getter* e *setter*.

2) Aggiungi alla classe Cliente la proprietà DataNascita (privata) definita come stringa con il formato "GG/MM/AAAA" nonché i relativi metodi *getter* e *setter* (quest'ultimo in grado di controllarne la validità).

3) Aggiungi alla classe Cliente la proprietà Comune (privata) di nascita definita come stringa nonché i relativi metodi *getter* e *setter*
(comune da scegliere in un determinato sottoinsieme - vedi tabella alla slide seguente**)**

4) Aggiungi alla classe Cliente la proprietà CodFiscale (privata), il metodo *getter* ed il metodo *setter* in grado di costruirlo e valorizzarlo nel seguente modo (lunghezza 15 caratteri):



12 TER - vedi Cliente-Multi-File-Overloading-Costruttore-Full.dev

Classe **Cliente**: **Estensione proprietà e metodi**

Mese	Cifre	Carattere
Gennaio	01	**A**
Febbraio	02	**B**
Aprile	03	**C**
Maggio	04	**D**
Maggio	05	**E**
Giugno	06	**H**
Luglio	07	**L**
Agosto	08	**M**
Settembre	09	**P**
Ottobre	10	**R**
Novembre	11	**S**
Dicembre	12	**T**

Comune	Codice
Napoli	**F839**
Pozzuoli	**G964**
Bacoli	**A535**
Monte di Procida	**F488**
Qualiano	**H101**
Quarto	**H114**

Esempio 1) **BIANCHI SERGIO 01/02/1950 Maschio Bacoli**
BIA SER 50 B 01 A535

Esempio 2) **ROSSI ELENA 23/09/1975 Femmina Pozzuoli**
ROS ELE 75 P 63 G964

Esempio 3) **VERDI SERENA 18/11/2003 Non definito Quarto**
VER SER 03 S 78 H114

Ovviamente questa non è la versione esatta del calcolo del codice fiscale…
Per chi fosse interessato è possibile consultare il seguente URL
https://www.studioaleo.it/struttura-codice-fiscale.html

Autore: Prof. Rio Chierego

(A) Classe **Moneta**: **un altro esempio svolto**

Si vuole effettuare la gestione di un programma che prevede un certo numeri di lanci di una **moneta** gestiti in **maniera random dal pc**.

La moneta può possedere, ovviamente in maniera schematica, due facce:
- una faccia principale che rappresent il valore **TESTA**;
- l'altra faccia che rappresenta il valore **CROCE**

L'utente, dopo avere scelto:
- quanti lanci effettuare
- **uno tra i valori possibili delle facce della moneta** (ossia TESTA o CROCE), assisterà all'esecuzione, da parte del pc, di tutti i lanci di moneta previsti.

Alla fine dei lanci riceverà l'esito finale del gioco sottoforma di messaggio a video secondo il seguente schema:
- se tra i lanci del pc è uscito più del 50% delle volte quanto da lui stabilito (TESTA o CROCE) allora stampare **"Tu HAI VINTO – IL pc HA PERSO!"**
- se tra i lanci del pc è uscito meno del 50% delle volte quanto da lui stabilito(TESTA o CROCE) allora stampare **"Tu HAI PERSO – IL pc HA VINTO!"**
- se tra i lanci del pc è uscito esattamente il 50% delle volte quanto da lui stabilito (TESTA o CROCE) allora stampare **"Tu ed il pc AVETE PAREGGIATO!"**

Dovrà quindi essere possibile **lanciare** la moneta, potendo poi **leggere** l'esito di tale lancio direttamente dalla sua **faccia** "virtuale" sottoforma delle stringhe "TESTA" e "CROCE"

(A) Progettazione: Esempio della classe **Moneta**

Esercizio: Proviamo ad implementare la classe **Moneta** che si occupa di effettuare il gioco descritto nella slide precedente

Moneta
- Faccia : **BOOL**
+ **FUNZIONE get_Faccia( ) : BOOL** + **PROCEDURA lancia( )** + **FUNZIONE toString(VAL f : BOOL) : string**

N.B. La proprietà privata **Faccia** è allocata STATICAMENTE

(A) Progettazione: Esempio della classe **Moneta**

Esercizio: implementiamo i **metodi** previsti per la classe

```
FUNZIONE get_Faccia ( ) : BOOL
INIZIO
RITORNA Faccia
FINE
```

```
FUNZIONE toString (VAL f : BOOL) : string
INIZIO
s : string
SE (f = VERO)
   ALLORA
      s ← "TESTA"
   ALTRIMENTI
      s ← "CROCE"
FINE SE
RITORNA s
FINE
```

```
PROCEDURA lancia ( )
INIZIO
f : INT
// Chiamata ad una funzione che generi casualmente
// il valore 0 oppure 1  (in C/C++ vedi srand() e rand()
f ← ..............
 SE (f = 1)
    ALLORA
        Faccia ← VERO;
    ALTRIMENTI
        Faccia ← FALSO;
FINE SE
RITORNA
FINE
```

(A) Progettazione: Esempio della classe **Moneta**

Esercizio: implementiamo ANCHE la PROCEDURA main( ) (parte 1/3)

```
ALGORITMO Testa_Croce
PROCEDURA main()
//Dichiarazione variabili inpuit/output/lavoro
esitoLancio : BOOL
esitoLancioS, risposta : string
i, nLanci, nWin : INT
// Dichiarazione oggetto della classe Moneta ALLOCATO STATICAMENTE
mycoin : Moneta

INIZIO
// Acquisisco il numero di lanci totali per il gioco
RIPETI
   Scrivi ("Inserisci il numero di lanci che vuoi effettuare: ")
   Leggi (nLanci)
FINCHE' (nLanci > 0)

// Leggo e controllo la scelta fatta dall'utente per il lancio della moneta
RIPETI
   Scrivi ("Fai la tua scelta (TESTA o CROCE): ")
   Leggi (risposta)
FINCHE' (risposta = "TESTA") OR (risposta = "CROCE")
```

(A) Progettazione: Esempio della classe **Moneta**

Esercizio: implementiamo la PROCEDURA main( ) (parte 2/3)

```
//Gestisco la serie di lanci stabilendo l'esito finale del gioco
nWin ← 0;
PER i ← 1 A nLanci ESEGUI
    // Effettuo il lancio della moneta
    mycoin.lancia();
    // Valuto l'esito del lancio e da booleano lo riporto in stringa
    esitoLancio ← mycoin.get_Testa()
    esitoLancioS ← mycoin.toString(esitoLancio)
    Scrivi ("E' uscito: ")
    Scrivi (esitoLancioS)
    //Scrivi (mycoin.toString(mycoin.get_Testa())
    // Controllo se l'esito del lancio della moneta coincide la scelta utente iniziale
    SE (esitoLancioS = risposta)
       ALLORA
           nWin ← nWin + 1
           Scrivi (" <--- beccato!")
    FINE SE
    i ← i + 1
FINE PER
```

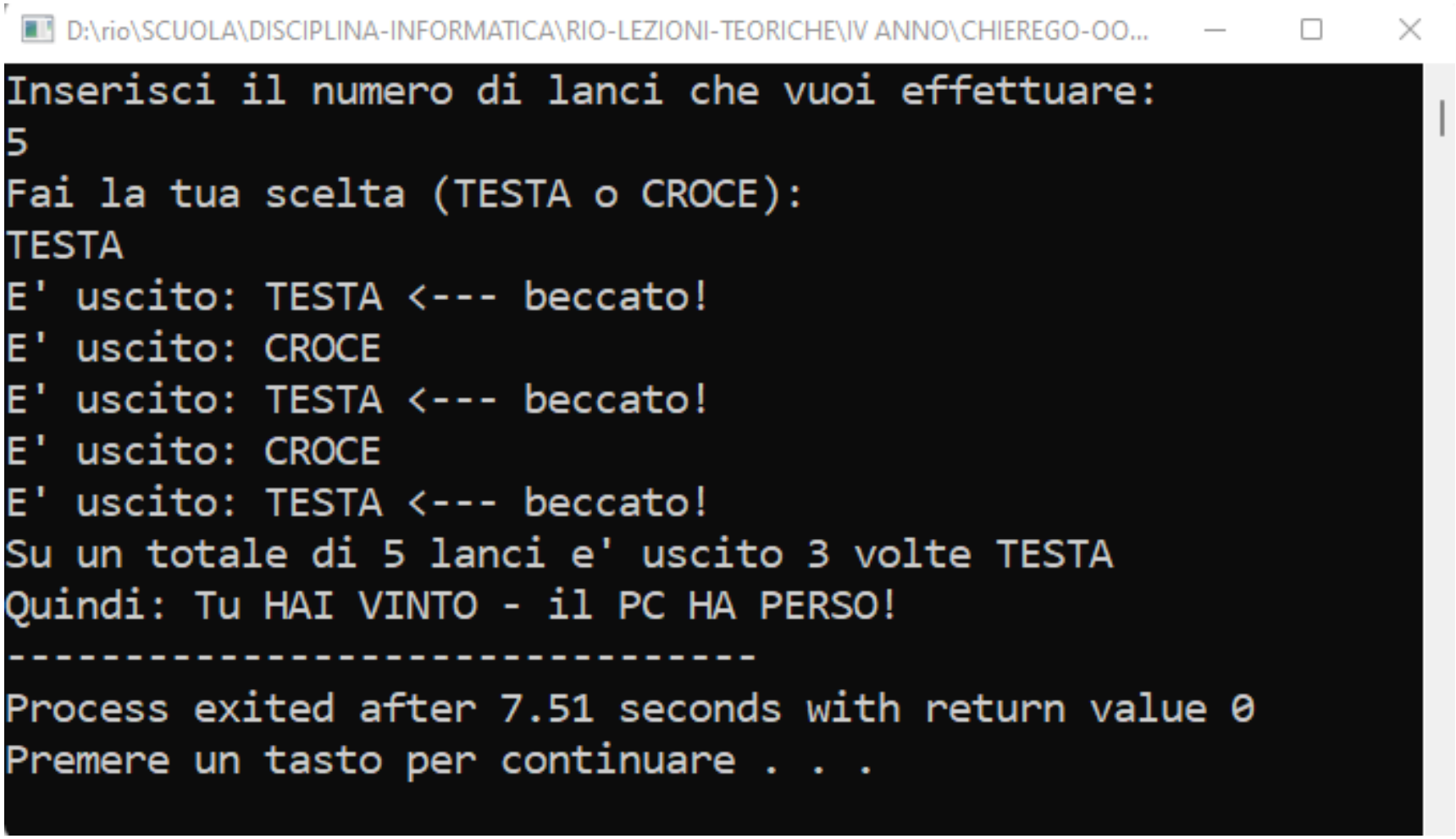
(A) Progettazione: Esempio della classe **Moneta**

Esercizio: implementiamo la PROCEDURA main( ) (parte 3/3)

```
// Valuto il totale delle vittorie (è uscito maggiormente quello che l'utente ha inizialmente indicato)
Scrivi ("Lanci totali coincidenti con la scelta fatta: ")
Scrivi (nWin)
// Ciò che ho scelto è uscito per piu' del 50% di volte
SE (nWin > nLanci - nWin)
   ALLORA
      Scrivi ("Quindi: Tu HAI VINTO - il PC HA PERSO!")
  ALTRIMENTI
     SE (nWin < nLanci - nWin)
        ALLORA
           // Ciò che ho scelto è uscito meno del 50% di volte
            Scrivi ("Quindi Tu HAI PERSO - Il PC HA VINTO!")
       ALTRIMENTI
            //Ciò che ho scelto è uscito esattamente pari al 50% di volte
            Scrivi ("Quindi Tu HAI PAREGGIATO con il PC!")
      FINE SE
  FINE SE
RITORNA
FINE
```

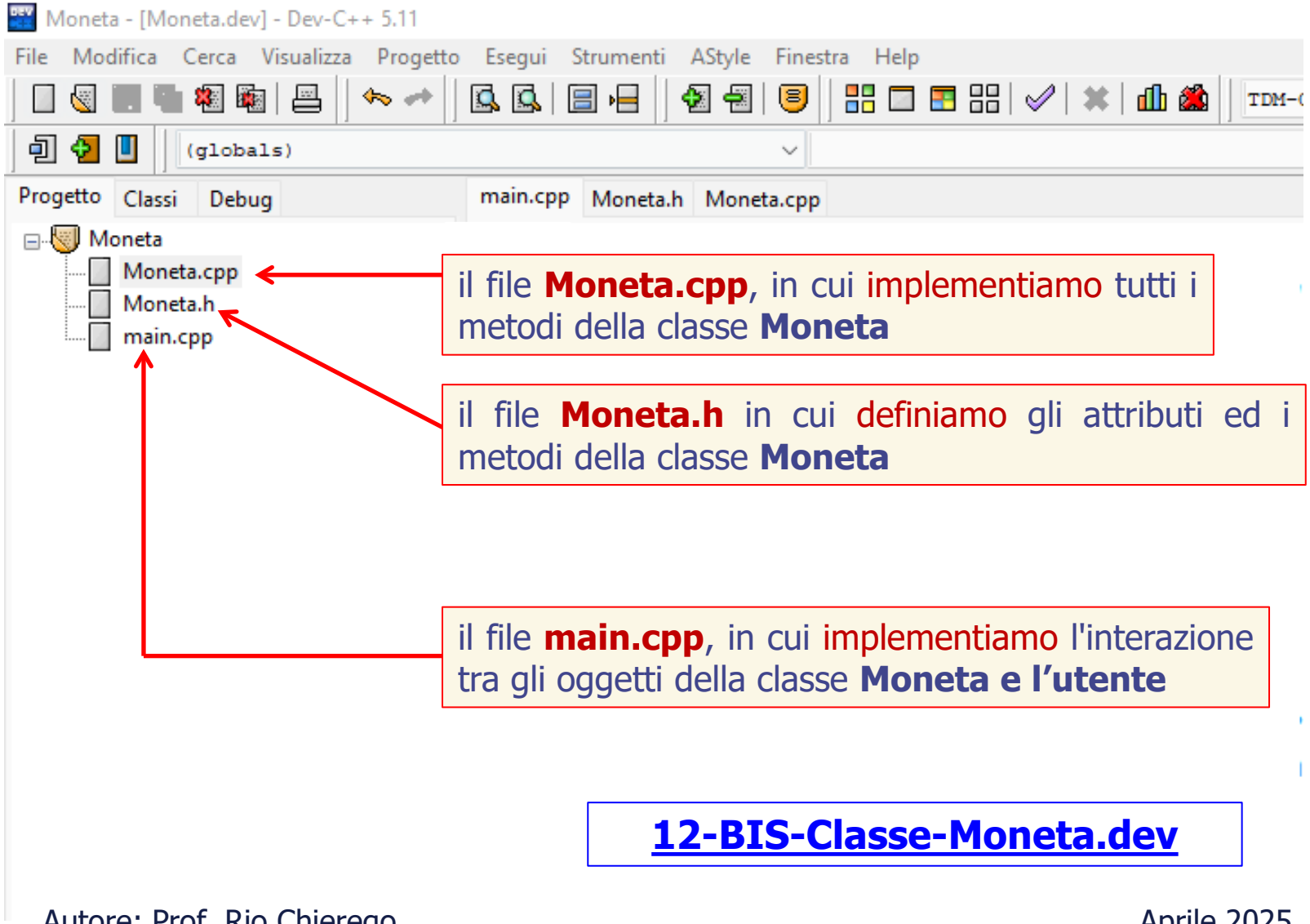
(A) Progettazione: Esempio della classe **Moneta**

Esempio di interazione: dopo la **codifica** avremo un'interazione con il programma di questo tipo



(A) Progettazione: Esempio della classe **Moneta**

Esercizio: Progetto **Moneta.dev** sviluppato su **tre file**



12-BIS-Classe-Moneta.dev

(A) Progettazione: Esempio della classe **Moneta**

File **Moneta.h**: implementiamo la classe **Moneta**

```
#ifndef _MONETA_H_
#define _MONETA_H_

#include <string>

using namespace std;

class Moneta
   {
   private:
      //Faccia moneta TESTA              --> faccia = VERO
      //Faccia moneta NON TESTA (CROCE) --> faccia = FALSO
      bool Faccia;
   public:
       //metodo pubblico che effettua il lancio della moneta
       void lancia (void);
       //metodo pubblico che restituisce la proprità privata faccia (metodo getter)
       bool get_Faccia (void);
       //metodo pubblico che converte un booleano nelle stringhe "TESTA" e "CROCE"
       string toString (bool);
   };

#endif
```

Nota Bene La direttiva **#ifndef** è dedicata alla fase di PRE-PROCESSIONE del codice ed è utile per verificare se un simbolo è già stato dichiarato ed evitare di dichiararlo più volte nello stesso codice sorgente.

La sintassi di #ifnfef
#ifndef [simbolo]
blocco di istruzioni
#endif

(A) Progettazione: Esempio della classe **Moneta**

File **Moneta.cpp**: implementiamo i metodi della classe **Moneta** (parte 1/2)

```
#include <ctime>
#include <stdlib.h>

#include "Moneta.h"

//metodo pubblico che si preoccupa di effettuare il lancio della moneta
void Moneta::lancia (void)
{
int f;

//genera casualmente il valore 0 oppure 1
f = rand() % 2;
if (f == 1)
    {
    Faccia = true;
    }
else
    {
    Faccia = false;
    }

return;
}
```

(A) Progettazione: Esempio della classe **Moneta**

File **Moneta.cpp**: implementiamo i metodi della classe **Moneta** (parte 2/2)

```
//metodo pubblico che restituisce il valore della proprità privata Faccia
//(metodo getter)
bool Moneta::get_Faccia(void)
{
return Faccia;
}

//metodo che converte i valori booleani nelle stringa "TESTA" o "CROCE"
string Moneta::toString (bool f)
{
string s;

if (f == true)
    {
    s = "TESTA";
    }
else
    {
    s = "CROCE";
    }

return s;
}
```

(A) Progettazione: Esempio della classe **Moneta**

File **main.cpp**: implementiamo il main effettuando un numero di lanci deciso dall'utente su di un oggetto della classe **Moneta** ALLOCATO STATICAMENTE **(parte 1/3)**

```
#include <iostream>

#include <ctime>
#include <stdlib.h>

// Include della classe Moneta
#include "Moneta.h"

using namespace std;

int main(int argc, char** argv)
{
bool esitoLancio;
string esitoLancioS, risposta;
int i, nLanci, nWin;

// Dichiarazione oggetto della classe Moneta ALLOCATO STATICAMENTE
Moneta mycoin;

// Inizializzo il generatore di numeri pseudo-casuali
srand(time(NULL));

// Acquisisco il numero di lanci totali per il gioco
do
   {
   cout << "Inserisci il numero di lanci che vuoi effettuare: " << endl;
   cin >> nLanci;
   }
while (nLanci <= 0);
```

(A) Progettazione: Esempio della classe **Moneta**

```
// Leggo e controllo la scelta fatta dall'utente per il lancio della moneta
do
   {
   cout << "Fai la tua scelta (TESTA o CROCE): " << endl;
   cin >> risposta;
   }
while ((risposta != "TESTA") && (risposta != "CROCE"));

//Gestisco la serie di lanci stabilendo l'esito finale del gioco
nWin = 0;
for (i = 1; i <= nLanci; i++)
   {
   // Effettuo il lancio della moneta
   mycoin.lancia();

   // Valuto l'esito del lancio e da booleano lo riporto in stringa
   esitoLancio = mycoin.get_Faccia();
   esitoLancioS = mycoin.toString(esitoLancio);
   cout << "E' uscito: " << esitoLancioS;
   //cout << "E' uscito: " << mycoin.toString(mycoin.get_Faccia());

   // Controllo se l'esito del lancio della moneta coincide la scelta utente iniziale
   if (esitoLancioS == risposta)
      {
      nWin++;
      cout << " <--- beccato!";
      }
   // Vado a capo
   cout << endl;
   }
```

File **main.cpp**: implementiamo il main effettuando un numero di lanci deciso dall'utente su di un oggetto della classe **Moneta** ALLOCATO STATICAMENTE **(parte 2/3)**

(A) Progettazione: Esempio della classe **Moneta**

File **main.cpp**: implementiamo il main effettuando un numero di lanci deciso dall'utente su di un oggetto della classe **Moneta** ALLOCATO STATICAMENTE **(parte 3/3)**

```
// Valuto il totale delle vittorie (è uscito maggiormente quello che
// l'utente ha inizialmente indicato)
cout << "Su un totale di " << nLanci << " lanci e' uscito ";
cout << nWin << " volte " << risposta << endl;
// Ciò che ho scelto è uscito per piu' del 50% di volte
if (nWin > nLanci - nWin)
   {
   cout << "Quindi: Tu HAI VINTO - il PC HA PERSO!";
   }
// Ciò che ho scelto è uscito meno del 50% di volte
else if (nWin < nLanci - nWin)
   {
   cout << "Quindi Tu HAI PERSO - Il PC HA VINTO!";
   }
// Ciò che ho scelto è uscito esattamente pari al 50% di volte
else
   {
   cout << "Quindi Tu HAI PAREGGIATO con il PC!";
   }

return 0;
}
```

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

Esercizio: Proviamo ad implementare la classe **Conversione** che si occupa di effettuare la conversione degli importi dalle lire all'euro e viceversa

Conversione	
- Lire : INT - Euro: REAL	
+ PROCEDURA set_Lire (VAL lit:INT) + FUNZIONE get_Lire() : INT + PROCEDURA set_Euro (VAL e:REAL) + FUNZIONE get_Euro() : REAL	
+ FUNZIONE converti(VAL lit : INT) : REAL	//funzione che converte le lire in euro
+ FUNZIONE converti(VAL e : REAL) : INT	//funzione che converte gli euro in lire
+ COSTRUTTORE Conversione( ) : Conversione	//Costruttore
+ DISTRUTTORE ~Conversione( )	// Distruttore

(B) Le proprietà private **Lire** ed **Euro** sono allocate STATICAMENTE

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

Esercizio: implementiamo i metodi previsti

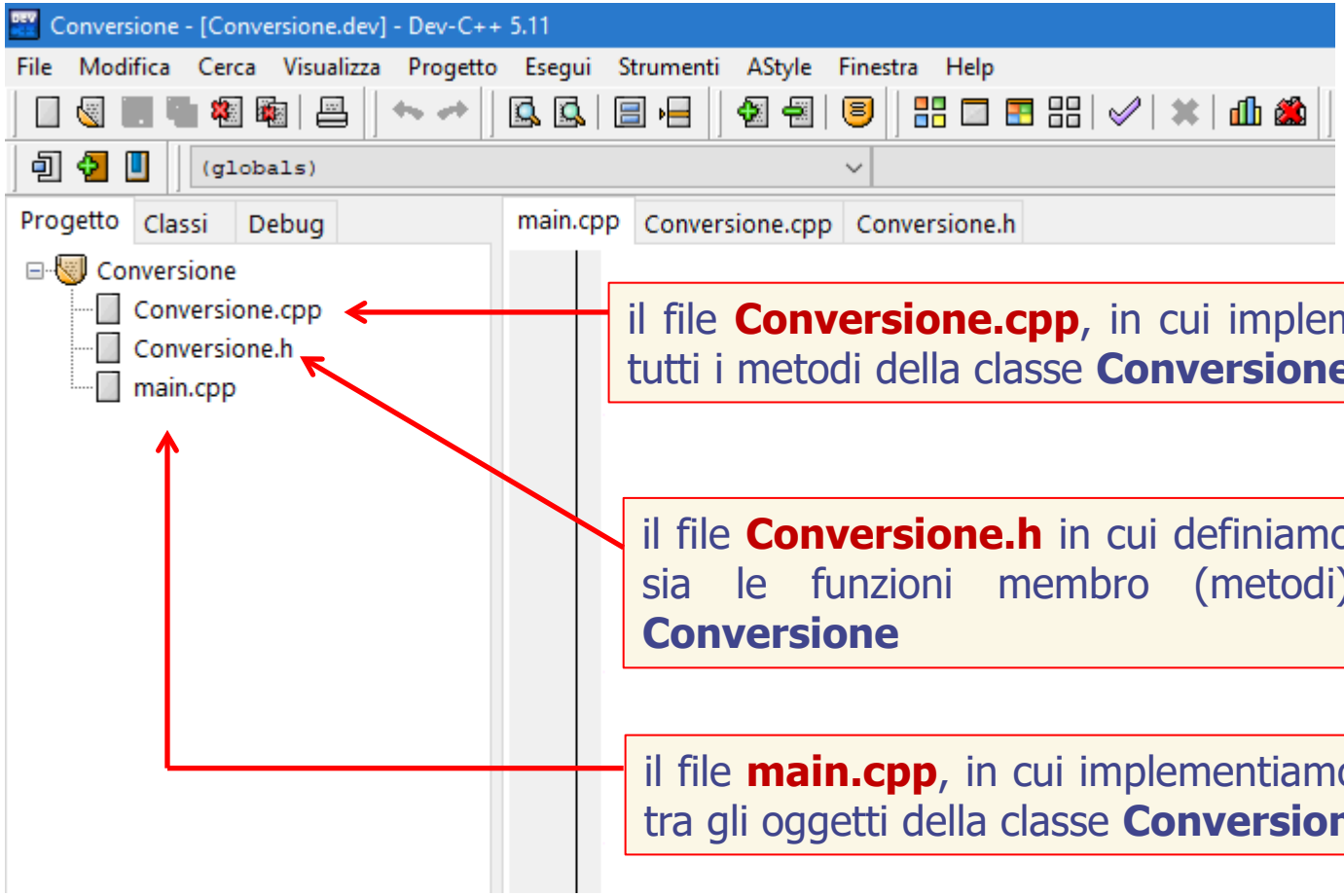
```
PROCEDURA set_Lire (VAL lit : INT)
INIZIO
Lire ←lit
RITORNA
FINE

FUNZIONE get_Lire () : INT
INIZIO
RITORNA Lire
FINE

PROCEDURA set_Euro (VAL e : REAL)
INIZIO
Euro ←e
RITORNA
FINE

FUNZIONE get_Euro ( ) : REAL
INIZIO
RITORNA Euro
FINE
```

```
FUNZIONE converti (VAL lit : INT) : REAL
INIZIO
set_Euro( lit / 1936.27)
RITORNA get_Euro( )
FINE

FUNZIONE converti (VAL e : REAL) : INT
INIZIO
set_Lire (e * 1936.27)
RITORNA get_Lire( )
FINE

COSTRUTTORE Conversione ( ) : Conversione
oggetto : Conversione
INIZIO
Scrivi("Costruttore: Inizio procedimento di conversione")
oggetto.Lire ← 0
oggetto.Euro ← 0.00
RITORNA oggetto
FINE
DISTRUTTORE ~Conversione ( )
INIZIO
Scrivi("Distruttore: Fine procedimento di conversione")
RITORNA
FINE
```

N.B. fare attenzione ed usare, ogni volta che è possibile, i metodi **setter** e **getter** creati proprio con questo scopo

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

Esercizio: Progetto **Conversione-LIT-EU-Statica.dev** sviluppato su **tre file**



il file **Conversione.cpp**, in cui implementiamo tutti i metodi della classe **Conversione**

il file **Conversione.h** in cui definiamo sia gli attributi sia le funzioni membro (metodi) della classe **Conversione**

il file **main.cpp**, in cui implementiamo l'interazione tra gli oggetti della classe **Conversione e l'utente**

13. vedi Conversione-LIT-EU-Statica.dev

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **Conversione.h**: implementiamo la classe **Conversione**

```
#include <iostream>
#include <cmath>

using namespace std;

//----- inizio dichiarazione classe Conversione
class Conversione
   {
   private:
      long Lire;
      float Euro;
   public:
      void set_Lire(long);
      long get_Lire();

      void set_Euro(float);
      float get_Euro();

      float converti(long);  //funzione che converte le lire in euro
      long converti(float);  //funzione che converte gli euro in lire

      Conversione();                 //Costruttore
      ~Conversione();                //Distruttore
   };
//------ fine dichiarazione classe Conversione
```

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **Conversione.cpp**: implementiamo i metodi della classe **Conversione** (parte 1/2)

```
#include "Conversione.h"

//----- inizio metodi PUBBLICI classe Conversione
void Conversione::set_Lire(long lit)
{
Lire = lit;
}

long Conversione::get_Lire()
{
return Lire;
}

void Conversione::set_Euro (float e)
{
Euro = e;
}

float Conversione::get_Euro()
{
return Euro;
}

Conversione::Conversione()
{
cout << "Costruttore: Inizio procedimento di conversione" << endl;
Lire = 0;
Euro = 0.00;
return;
}

Conversione::~Conversione()
{
cout << "Distruttore: Fine del procedimento di conversione" << endl;
return;
}
```

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **Conversione.cpp**: implementiamo i metodi della classe **Conversione** (parte 2/2)

```
float Conversione::converti(long lit)
{
float f;
char s[20];

//calcolo il valore in euro relativo all'importo in lire inserito
f = (float) lit / 1936.27;

//risolvo il problema dell'approssimazione ai decimali tramite la funzione C sprintf
sprintf(s, "%.2f", f);

//valorizzo la proprietà Euro della classe
set_Euro(atof(s));

return get_Euro();
}

long Conversione::converti(float e)
{
long lit;

//calcolo il valore in lire relativo all'importo in euro inserito
lit = round(e * 1936.27);

//valorizzo la proprietà Lire della classe
set_Lire(lit);

return get_Lire();
}
```

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **main.cpp**: implementiamo il main effettuando delle conversioni tramite un oggetto della classe **Conversione** ALLOCATO STATICAMENTE (parte 1/2)

```
#include "Conversione.h"

//----- inizio main
int main(int argc, char** argv)
{
Conversione c;
long lit;
float e;

//LIRE ---> EURO
cout << "MAIN: Inserisci le LIRE: ";
fflush(stdin);
cin >> lit;
c.set_Lire(lit);

//conversione LIRE-EURO
e = c.converti(lit);
cout << "MAIN: Dopo conversione EURO = " << e << endl;

//Stampa LIRE - EURO dell'oggetto corrente c della classe Conversione
cout << "MAIN: valorizzazione dell'oggetto corrente c della classe Conversione" << endl;
cout << "LIRE = " << c.get_Lire() << endl;
cout << "EURO = " << c.get_Euro() << endl;
```

L'oggetto **c** della classe **Conversione** è stato **ALLOCATO STATICAMENTE ....**

... e quindi per accedere ai suoi **metodi/proprietà** occorrerà utilizzare **l'operatore punto** ossia **.**

(B1) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **main.cpp**: implementiamo il main effettuando delle conversioni tramite un oggetto della classe **Conversione** ALLOCATO STATICAMENTE (parte 2/2)

```
//EURO ---> LIRE
cout << "MAIN: Inserisci gli EURO (2 decimali): ";
fflush(stdin);
cin >> e;
c.set_Euro(e);

//conversione EURO-LIRE
lit = c.converti(e);
cout << "MAIN: Dopo conversione LIRE = " << lit << endl;

//Stampa EURO-LIRE dell'oggetto corrente c della classe Conversione
cout << "MAIN: valorizzazione dell'oggetto corrente c della classe Conversione" << endl;
cout << "EURO = " << c.get_Euro() << endl;
cout << "LIRE = " << c.get_Lire() << endl;

return 0;
}
//----- fine main
```

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

Esercizio: Proviamo ad implementare la classe **Conversione** che si occupa di effettuare la conversione degli importi dalle lire all'euro e viceversa

Conversione
- Lire : PUNTATORE A INT - Euro: PUNTATORE A REAL
+ PROCEDURA set_Lire (VAL lit:INT) + FUNZIONE get_Lire() : INT + PROCEDURA set_Euro (VAL e:REAL) + FUNZIONE get_Euro() : REAL + FUNZIONE converti(VAL lit : INT) : REAL //funzione che converte le lire in euro + FUNZIONE converti(VAL e : REAL) : INT //funzione che converte gli euro in lire + COSTRUTTORE Conversione( ) : Conversione //Costruttore + DISTRUTTORE ~Conversione( ) // Distruttore

(B) Le proprietà Lire ed Euro sono allocate DINAMICAMENTE

14. vedi Conversione-LIT-EU-Dinamica.dev

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

Esercizio: implementiamo i metodi previsti

```
PROCEDURA set_Lire (VAL lit : INT)
INIZIO
*Lire ←lit
RITORNA
FINE

FUNZIONE get_Lire () : INT
INIZIO
RITORNA *Lire
FINE

PROCEDURA set_Euro (VAL e : REAL)
INIZIO
*Euro ←e
RITORNA

FINE
FUNZIONE get_Euro ( ) : REAL
INIZIO
RITORNA *Euro
FINE

FUNZIONE converti (VAL lit : INT) : REAL
INIZIO
set_Euro( lit / 1936.27)
RITORNA get_Euro( )
FINE
```

```
FUNZIONE converti (VAL e : REAL) : INT
INIZIO
set_Lire (e * 1936.27)
RITORNA get_Lire( )
FINE

COSTRUTTORE Conversione ( ) : Conversione
oggetto : Conversione
INIZIO
Scrivi("Costruttore: Inizio procedimento di conversione")
Alloca(Lire, DimensioneDi(INT))
*(oggetto.Lire) ← 0
Alloca(Euro, Dimensione>Di(REAL))
*(oggetto.Euro) ← 0.00
RITORNA oggetto
FINE

DISTRUTTORE ~Conversione ( )
INIZIO
Scrivi("Distruttore: Fine procedimento di conversione")
Dealloca(Lire)
Dealloca(Euro)
RITORNA
FINE
```

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **Conversione.h**: implementiamo la classe **Conversione**

```
#include <iostream>
#include <cmath>

using namespace std;

//----- inizio dichiarazione classe Conversione
class Conversione
   {
   private:
      long* Lire;          //pointer al tipo long int
      float* Euro;         //pointer al tipo float
   public:
      void set_Lire(long);
      long get_Lire();

      void set_Euro(float);
      float get_Euro();

      float converti(long);  //funzione che converte le lire in euro
      long converti(float);  //funzione che converte gli euro in lire

      void converti(long, float&);

      Conversione();                //Costruttore
      ~Conversione();               //Distruttore
   };
//------ fine dichiarazione classe Conversione
```

Le variabili **Lire** e **Euro** sono state definite ora come puntatori. Perciò sarà necessario allocare spazio per utilizzarle e ciò, come buona norma insegna, viene effettuato nel **costruttore** della classe

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **Conversione.cpp**: implementiamo i metodi della classe **Conversione** (parte 1/2)

```
#include "Conversione.h"

//----- inizio metodi PUBBLICI classe Conversione
void Conversione::set_Lire(long lit)
{
*Lire = lit;
}

long Conversione::get_Lire()
{
return *Lire;
}

void Conversione::set_Euro (float e)
{
*Euro = e;
}

float Conversione::get_Euro()
{
return *Euro;
}

Conversione::Conversione()
{
cout << "Costruttore: Inizio procedimento di conversione" << endl;
Lire = new long (0);
Euro = new float(0.00);
return;
}

Conversione::~Conversione()
{
cout << "Distruttore: Fine del procedimento di conversione" << endl;
delete(Lire);
delete(Euro);
return;
}
```

Allocazione dinamica dello spazio necessario per le variabili **Lire** e **Euro** nel **costruttore** della classe

Deallocazione dello spazio per le variabili **Lire** e **Euro** nel **distruttore** della classe

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **Conversione.cpp**: implementiamo i metodi della classe **Conversione** (parte 2/2)

```
float Conversione::converti(long lit)
{
float f;
char s[20];

//calcolo il valore in euro relativo all'importo in lire inserito
f = (float) lit / 1936.27;

//risolvo il problema dell'approssimazione ai decimali tramite la funzione C sprintf
sprintf(s, "%.2f", f);

//valorizzo la proprietà Euro della classe tramite punatore this (oggetto corrente)
*(this->Euro) = atof(s);

return *(this->Euro);
}

long Conversione::converti(float e)
{
long lit;

//calcolo il valore in lire relativo all'importo in euro inserito
lit = round(e * 1936.27);

//valorizzo la proprietà Lire della classe tramite punatote this (oggetto corrente)
*(this->Lire)= lit;

return lit;
}
//----- fine metodi PUBBLICI classe Conversione
```

Utilizzo l'operatore * applicato alla variabile **Euro** di tipo **puntatore a float** per fornire in valore in euro convertito.

Utilizzo l'operatore * applicato alla variabile **Lire** di tipo **puntatore a long** per fornire in valore in euro convertito.

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **main.cpp**: implementiamo il main effettuando delle conversioni tramite un oggetto della classe **Conversione** ALLOCATO DINAMICAMENTE(parte 1/2)

```
#include "Conversione.h"

//----- inizio main
int main(int argc, char** argv)
{
long lit;
float e;
//allocazione dinamica dell'oggetto della classe Conversione nello heap
Conversione* c = new Conversione();

//LIRE ---> EURO
cout << "MAIN: Inserisci le LIRE: ";
fflush(stdin);
cin >> lit;
c->set_Lire(lit);

//conversione LIRE-EURO
e = c->converti(lit);
cout << "MAIN: Dopo conversione EURO = " << e << endl;

//Stampa LIRE - EURO dell'oggetto corrente c della classe Conversione
cout << "MAIN: valorizzazione dell'oggetto corrente c della classe Conversione" << endl;
cout << "LIRE = " << c->get_Lire() << endl;
cout << "EURO = " << c->get_Euro() << endl;
```

L'oggetto **c** della classe **Conversione** è stato **ALLOCATO DINAMICAMENTE** tramite la funzione **new()**

Dunque per accedere ai suoi **metodi/proprietà** occorrerà utilizzare **l'operatore freccia** ossia **->**

(B2) Progettazione: Esempio di classe per la conversione LIRE-EURO

File **main.cpp**: implementiamo il main effettuando delle conversioni tramite un oggetto della classe **Conversione** ALLOCATO STATICAMENTE (parte 2/2)

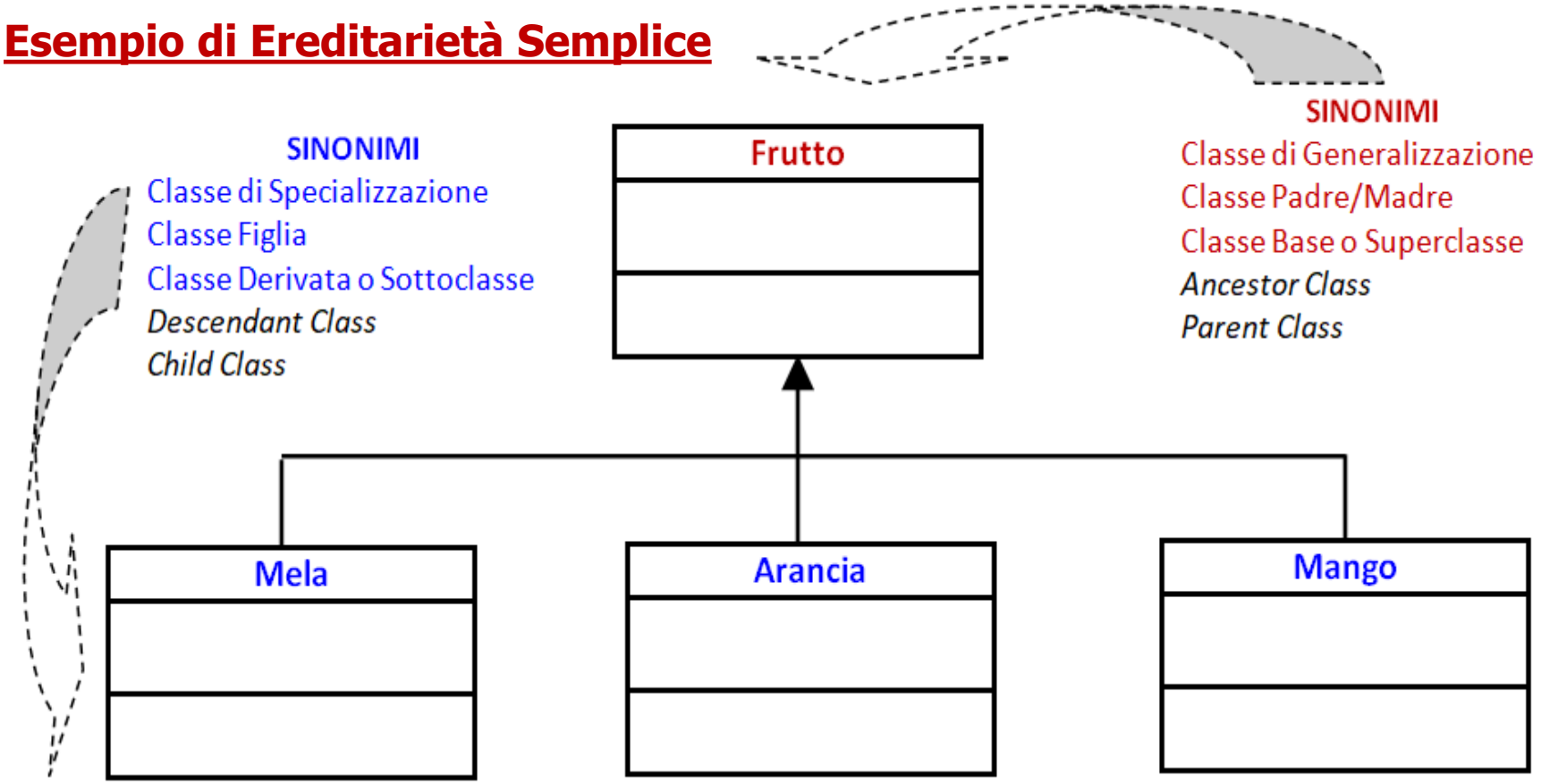
```
//EURO ---> LIRE
cout << "MAIN: Inserisci gli EURO (2 decimali): ";
fflush(stdin);
cin >> e;
c->set_Euro(e);

//conversione EURO-LIRE
lit = c->converti(e);
cout << "MAIN: Dopo conversione LIRE = " << lit << endl;

//Stampa EURO-LIRE dell'oggetto corrente c della classe Conversione
cout << "MAIN: valorizzazione dell'oggetto corrente c della classe Conversione" << endl;
cout << "EURO = " << c->get_Euro() << endl;
cout << "LIRE = " << c->get_Lire() << endl;

//liberazione area di memoria allocata nello heap per l'oggetto della classe
delete(c);

return 0;
}
//----- fine main
```

Linguaggio C++: Ereditarietà

L'ereditarietà è il concetto alla base del meccanismo di **derivazione.**

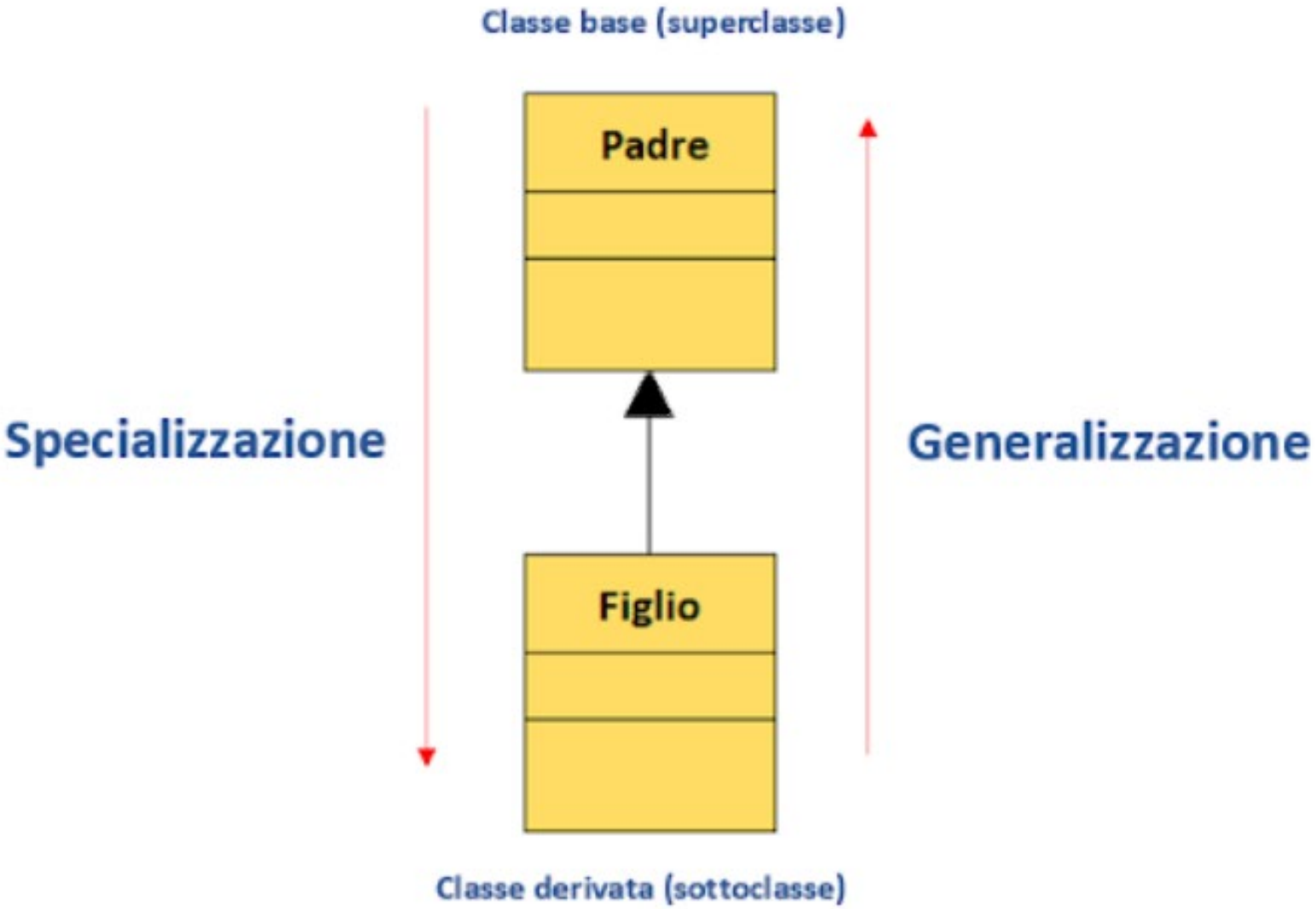
Derivazione: meccanismo attraverso il quale è possibile generare nuove classi a partire da una classe (**ereditarietà semplice**) o da più classi (**ereditarietà multipla**) **già esistenti**. N.B. La freccia indica la/e classe/i dalla quale si discende)

Esempio di Ereditarietà Semplice



Linguaggio C++: Ereditarietà

NOTA BENE: notazione di un'ISA



Linguaggio C++: Ereditarietà (semplice)

Una classe derivata (CHILD) eredita tutte le proprietà/metodi della classe base (PARENT)

In questo modo, è facile intuire, che la costruzione di classi complesse diventa sicuramente più semplice perché si risparmia buona parte del lavoro.

E' sufficiente infatti scegliere una classe genitore il più possibile simile alla classe figlia che si vuole realizzare per poi aggiungere a quest'ultima le nuove proprietà/metodi desiderati (**estensione**) e nel caso modificare (**overload**) oppure lasciare inalterate le segnature (**override**) dei metodi ricevuti in eredità

Un'altra valida ragione per usare quando possibile l'ereditarietà è la possibilità di riutilizzare facilmente il codice di un progetto precedente (**tecnica del RIUSO**) o più semplicemente, quando è necessario, apportare delle modifiche a tale progetto con un dispendio minimo di energie

Linguaggio C++: Ereditarietà (semplice)

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 (semplice)

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 (semplice)

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 (semplice)

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++: polimorfismo – OVERLOAD

Abbiamo già visto nelle slide precedenti che **l'overloading** delle funzioni è una caratteristica specifica del linguaggio C++ che non è presente nel linguaggio C e che può essere usata **anche senza utilizzare le classi**

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

L'overload 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 grazie al **binding dinamico**

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

Linguaggio C++: polimorfismo – OVERRIDE

Il **polimorfismo** nel linguaggio C++ si realizza oltre che con l'overloading delle funzioni (metodi) anche tramite l'**overriding** delle funzioni (metodi)

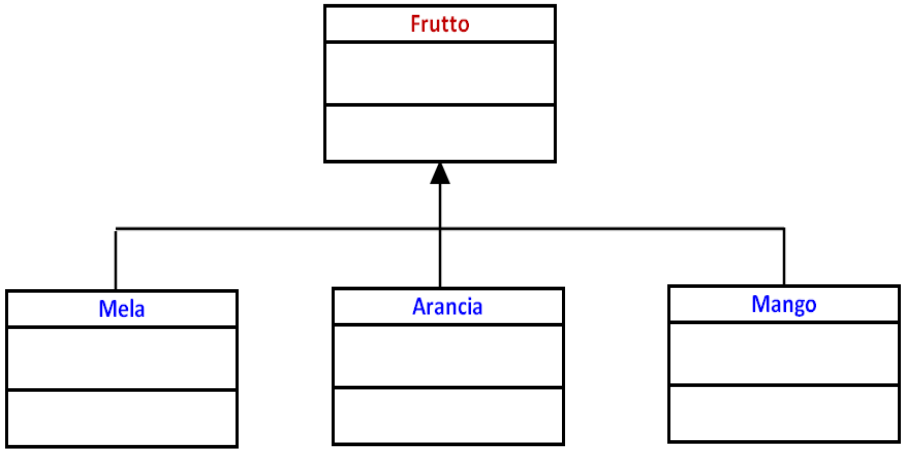
Con il termire **override** si intende una vera e propria riscrittura all'interno di una **classe derivata** (di specializzazione o figlia o **CHILD**) della logica risolutiva di un certo metodo che è stato ereditato da una **classe base** (di generalizzazione o padre o **PARENT**)

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

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

Ereditarietà: Cambio di classe per un oggetto (CASTING)

Supponiamo di avere due oggetti, x un Frutto ed y una Mela appartenenti a questa gerarchia di classi utilizzata più volte nelle nostre slide.



Supponiamo di avere effettuato le due dichiarazioni seguenti:

Frutto x;

Mela y;

Cosa possiamo dire delle assegnazioni

1) x = y; // OK CASTING implicito

2) y = x; // NOT OK: PROBLEMI!!

Un oggetto di una classe può diventare un oggetto di un'altra classe?

La prima istruzione è più immediata, perché se **y** è una **Mela** è sicuramente un **Frutto** (parliamo in questo caso di **CASTING implicito**)

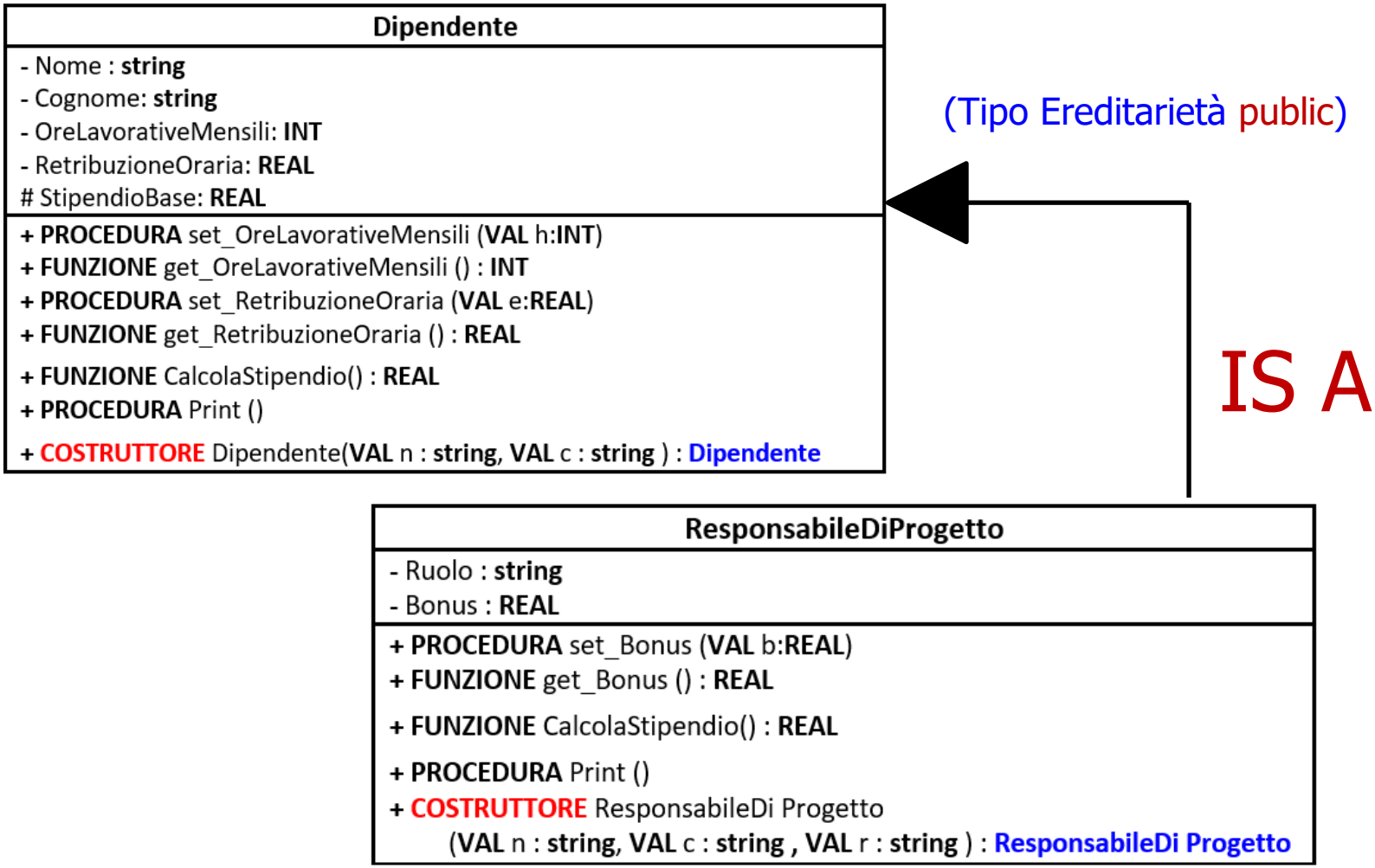
Non è così scontata la seconda perché se **x** è un **Frutto** potrebbe non essere una **Mela** bensì un'**Arancia** o un **Mango**. In questo caso è possibile effettuare un **CASTING esplicito** ossia un'operazione che forza un oggetto di una classe a diventare oggetto di un'altra classe a **PATTO CHE ESSE SIANO SULLO STESSO RAMO DELLA GERARCHIA**. E' corretto allora scrivere così:

2) y = (Mela) x; // OK: necessario CASTING ESPLICITO

Linguaggio C++: Ereditarietà semplice – Un esempio svolto

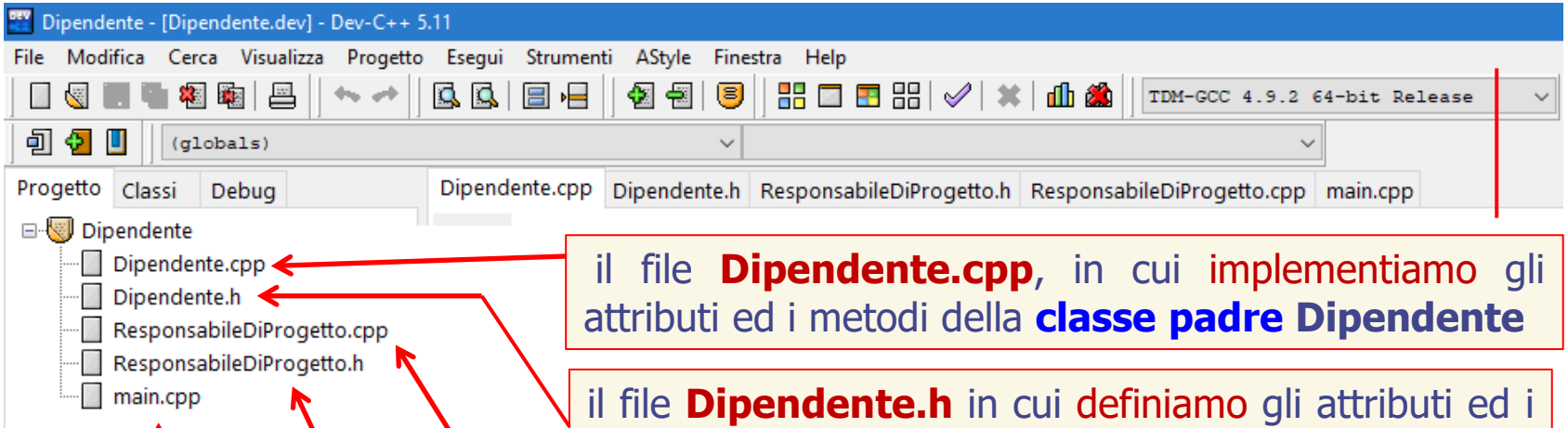
Inheritance Relationship o relazione di generalizzazione-specializzazione

16. vedi Dipendente-Inheritance.dev



Linguaggio C++: Ereditarietà semplice – Un esempio svolto

Esercizio: Progetto **Dipendente-Inheritance.dev** sviluppato su **cinque file**



il file **Dipendente.cpp**, in cui implementiamo gli attributi ed i metodi della **classe padre Dipendente**

il file **Dipendente.h** in cui definiamo gli attributi ed i metodi della **classe padre Dipendente**

il file **ResponsabileDiProgetto.cpp** in cui implementiamo gli attributi ed i metodi della **classe figlia ResponsabileDiProgetto** derivata da **Dipendente**

il file **ResponsabileDiProgetto.h** in cui definiamo gli attributi ed i metodi della **classe figlia ResponsabileDiProgetto**

il file **main.cpp**, in cui gestiremo un oggetto della **classe figlia ResponsabileDiProgetto**

Linguaggio C++: Ereditarietà semplice – Un esempio svolto

Per capire il significato vero e proprio del meccanismo di **OVERRIDE** di un metodo implementiamo l'esempio basato sulla classe "Dipendente" prima progettata
(N.B. Sono stati omessi i metodi ***setter*** e ***getter*** per gli attributi "nome" e "cognome" prevedendo un COSTRUTTORE ad hoc che li accetti come parametri)

File **Dipendente.h**

```
#include <iostream>
using namespace std;
//----- inizio definizione classe base Dipendente
class Dipendente
{
private:
        string Nome;
        string Cognome;
        int OreLavorativeMensili;
        float RetribuzioneOraria;
protected:
        float StipendioBase;
public:
        //METODI setter e getter proprietà private
        void set_OreLavorativeMensili(int h);
        int get_OreLavorativeMensili();
        void set_RetribuzioneOraria(float e);
        float get_RetribuzioneOraria();
        //N.B. METODO CHE SUBIRA' L'OVERRIDING NELLA CLASSE DERIVATA
        float CalcolaStipendio();
        //N.B. METODO CHE SUBIRA' L'OVERRIDING NELLA CLASSE DERIVATA
        void Print (void);
        //METODO COSTRUTTORE con parametri
        Dipendente (string n, string c);
};
//----- fine definizione classe base Dipendente
```

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

N.B. metodi che subiranno **l'OVERRIDE** nella **classe figlia ResponsabileDiProgetto**

Linguaggio C++: Ereditarietà semplice – Un esempio svolto

File **Dipendente.cpp (1/2)**

```
#include "Dipendente.h"
//----- inizio definizione METODI classe base Dipendente
void Dipendente::set_OreLavorativeMensili(int h)
{
OreLavorativeMensili = h;
return;
}

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

void Dipendente::set_RetribuzioneOraria (float e)
{
RetribuzioneOraria = e;
return;
}

float Dipendente::get_RetribuzioneOraria ()
{
return RetribuzioneOraria;
}

float Dipendente::CalcolaStipendio()
{
StipendioBase = get_OreLavorativeMensili() * get_RetribuzioneOraria();
return StipendioBase;
}
```

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

Linguaggio C++: Ereditarietà semplice – Un esempio svolto

N.B. il metodo stampa i dati privati di una qualunque istanza (oggetto) appartenente alla classe **Dipendente**

File **Dipendente.cpp (2/2)**

```
void Dipendente::Print (void)
{
cout << endl << "----------------------" << endl;
cout << "Dati Dipendente" << endl;
cout << "----------------------" << endl;
cout << "Nome: " << Nome << endl;
cout << "Cognome: " << Cognome << endl;
return;
}

Dipendente::Dipendente (string n, string c)
{
Nome = n;
Cognome = c;
return;
}
```

N.B. il metodo costruttore della **classe Dipendente** inizializzerà le proprietà private "nome" e "cognome"

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

File **ResponsabileDiProgetto.h**

```
#include "Dipendente.h"
//----- inizio definizione classe derivata ResponsabileDiProgetto
class ResponsabileDiProgetto: public Dipendente
{
private:
        string Ruolo;
        float Bonus;
public:
        //METODI setter e getter proprietà privata
        void set_Bonus(float b);
        float get_Bonus();
        //N.B. OVERRIDE DEL METODO DELLA CLASSE BASE
        float CalcolaStipendio();
        //N.B. OVERRIDE DEL METODO DELLA CLASSE BASE
        void Print (void);
        //METODO COSTRUTTORE con parametri (i primi due per classe dipendente
        ResponsabileDiProgetto (string n, string c, string r);
};
```

N.B. i metodi **CalcolaStipendio ( )** e **Print ( )** della **classe figlia ResponsabileDiProgetto hanno la stessa segnatura** degli omonimi metodi della **classe padre Dipendente (OVERRIDE)**

Il costruttore della **classe figlia ResponsabileDiProgetto** utilizzerà i primi due parametri per valorizzare le proprietà private "nome" e "cognome" della **classe padre Dipendente**

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

File **ResponsabileDiProgetto.cpp (1/2)**

```
#include "ResponsabileDiProgetto.h"
//----- inizio definizione METODI classe derivata ResponsabileDiProgetto
void ResponsabileDiProgetto::set_Bonus(float b)
{
Bonus = b;
return;
}

float ResponsabileDiProgetto::get_Bonus()
{
return Bonus;
}

//METODO che ha subito l'OVERRIDE
float ResponsabileDiProgetto::CalcolaStipendio()
{
float s;
s = (get_OreLavorativeMensili() * get_RetribuzioneOraria()) + Bonus;
return s;
}
```

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

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

File **ResponsabileDiProgetto.cpp (2/2)**

```
//METODO che ha subito l'OVERRIDE
void ResponsabileDiProgetto::Print (void)
{
//Invocazione analogo metodo classe PADRE Dipendente
Dipendente::Print();

cout << "----------------------" << endl;
cout << "Dati Responsabile di Progetto" << endl;
cout << "----------------------" << endl;
cout << "Ruolo: " << Ruolo << endl;
cout << "Bonus: " << Bonus << endl;
cout << "Ore lavorative mensili: " << this->get_OreLavorativeMensili() << endl;
cout << "Retribuzione oraria: " << this->get_RetribuzioneOraria() << endl;
return;
}

//METODO COSTRUTTORE dhe sfrutta il metodo costruttore della classe PADRE
ResponsabileDiProgetto::ResponsabileDiProgetto (string n, string c, string r) : Dipendente (n, c)
{
Ruolo = r;
return;
}
```

N.B. il metodo **Print ( )** si specializza tramite OVERRIDE per ogni istanza (oggetto) della **classe figlia ResponsabileDiProgetto** aggiungendo i dati della **classe padre Dipendente**

Il costruttore della **classe derivata ResponsabileDiProgetto** invoca il costruttore della **superclasse Dipendente** passandogli i primi due parametri per "Nome" e "Cognome"

Linguaggio C++: Ereditarietà semplice – Un esempio svolto

```
#include "ResponsabileDiProgetto.h"
//----- inizio main
int main(int argc, char** argv)
{
int h;
float e, b;

//Definizione dell'oggetto ResponsabileDi Progetto con costruttore con parametri
ResponsabileDiProgetto r = ResponsabileDiProgetto ("Gianni", "Verdi", "Direttore");

cout << "Inserire le Ore Lavorative Mensili del responsabile: ";
fflush(stdin);
cin >> h;
r.set_OreLavorativeMensili(h);

cout << "Inserire la Retribuzione Oraria del responsabile: ";
fflush(stdin);
cin >> e;
r.set_RetribuzioneOraria(e);

cout << "Inserire il bonus del responsabile: ";
fflush(stdin);
cin >> b;
r.set_Bonus(b);

//Invocazione metodo Print() sul ResponsabileDi Progetto
r.Print();

cout << "**********************" << endl;
cout << "Stipendio CALCOLATO del responsabile: ";
cout << r.CalcolaStipendio() << endl;

return 0;
}
```

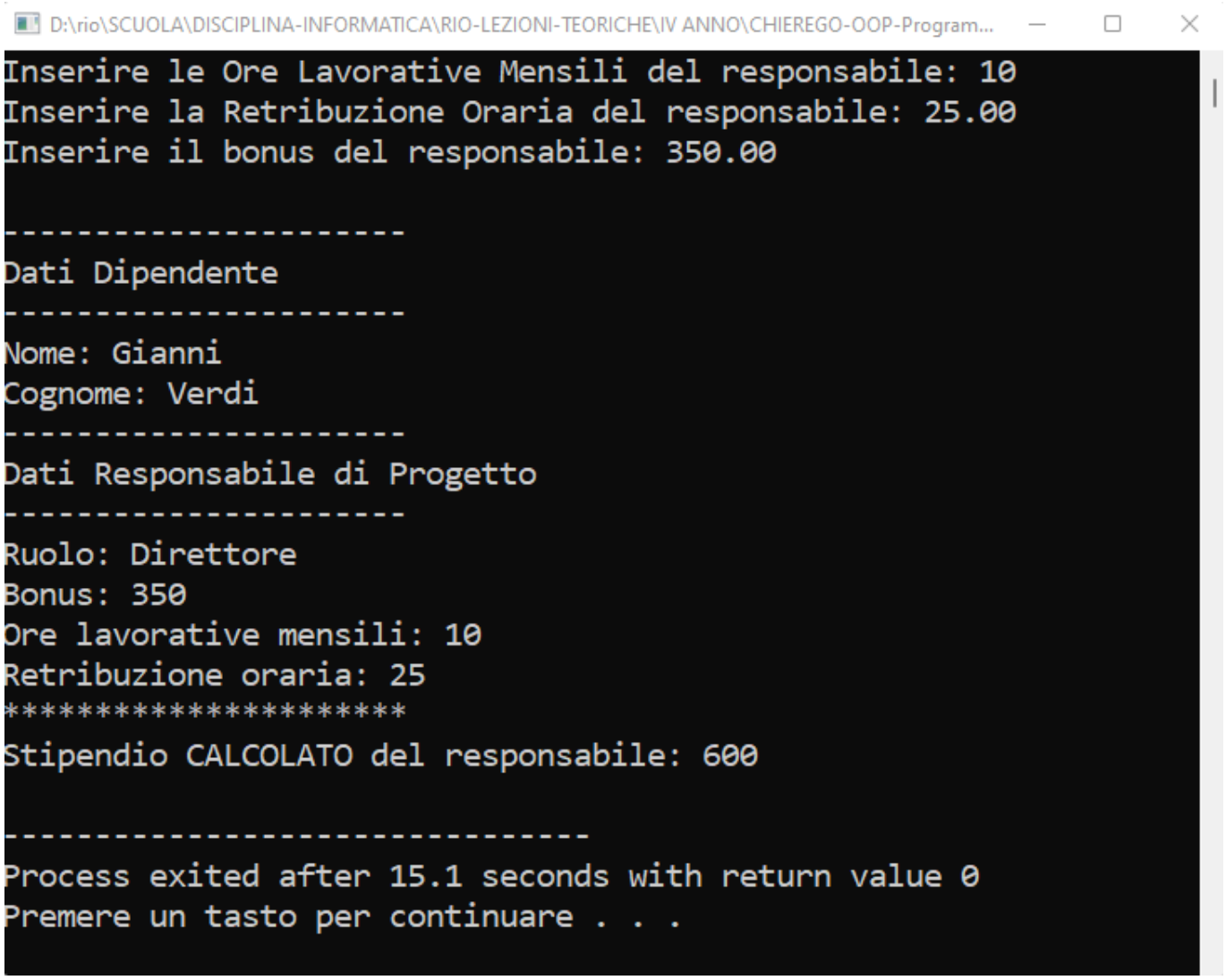
N.B. Viene definito un oggetto della **classe figlia ResponsabileDiProgetto** attraverso il costruttore implementato con parametri

File **main.cpp**

N.B. Vengono attivati i metodi **calcolaStipendio ( )** e **Print ( )** della **classe figlia** ottenuti applicando **l'OVERRIDE** ai metodi omonimi della **classe padre Dipendente**

Linguaggio C++: Ereditarietà semplice – Un esempio svolto

Esempio di interazione: dopo la **codifica** avremo un'interazione con il programma di questo tipo



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 e sidecar

Tutti i veicoli derivano 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

17. vedi Veicolo-Funzioni-Virtual.dev

Linguaggio C++: le funzioni virtuali (esempio)

```
#include <iostream>

using namespace std;

//----- classe Veicolo (PARENT)
 class Veicolo
 {
 public:
      virtual void numRuote()
      {
      return;
      };
 };

//----- classe Automobile (CHILD)
class Automobile: public Veicolo
 {
 public:
       void numRuote();
 };

//----- classe Automobile: OVERRIDE funzione virtuale numRuote() della classe PARENT
void Automobile::numRuote()
{
cout << "Un'Automobile IS 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. Il metodo poteva essere scritto anche così:

virtual void numRuote () { };

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)

```
//----- classe Moto (CHILD)
class Moto: public Veicolo
 {
 public:
       void numRuote();
 };

//----- classe Moto: OVERRIDE funzione virtuale numRuote() della classe PARENT
void Moto::numRuote()
{
cout << "Una Moto IS A Veicolo ed ha 2 ruote" << endl;
}

//----- classe Sidecar (CHILD)
class Sidecar: public Veicolo
 {
 public:
       void numRuote();
 };

//----- classe Sidecar: OVERRIDE funzione virtuale numRuote() della classe PARENT
void Sidecar::numRuote()
{
cout << "Un Sidecar IS 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)

Binding dinamico

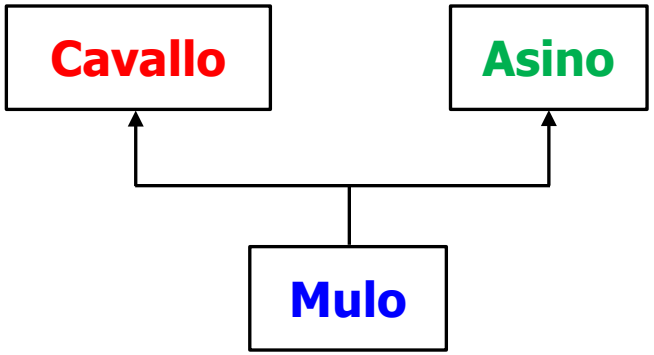
Quando viene ereditata una funzione virtuale viene ereditata anche la sua natura virtuale.
Questo significa che quando una classe derivata che abbia ereditato una funzione virtuale viene a sua volta utilizzata come classe base per un ulteriore classe derivata la funzione resta virtuale.

Linguaggio C++: Ereditarietà (multipla)

Torniamo ora al concetto di **ereditarietà**.
Dell'**ereditarietà semplice** abbiamo già parlato nelle slide precedenti ma ora usiamo un esempio per specificare il concetto di **ereditarietà multipla**.

Supponendo di voler spiegare a qualcuno che non lo sapesse cosa è un **mulo**, il modo più semplice è descriverlo come l'animale che deriva da un incrocio tra un cavallo con un asina e che eredita le caratteristiche di entrambi.

In questo modo si riuscirà a fornire una descrizione sufficientemente chiara senza perdersi nel dettaglio dei singoli particolari.



Una delle più potenti innovazioni apportate dal linguaggio C++ è proprio **l'ereditarietà multipla,** che consiste nella possibilità di derivare un oggetto da più classi.

Linguaggio C++: Ereditarietà (multipla)

La sintassi usata nel lingauggio C++ per rappresentare **l'ereditarietà multipla** applicata al caso in esame è una naturale estensione dell'ereditarietà semplice.

Nel nostro caso:

```
class Mulo : public Cavallo, public Asino
  {
  ......
  };
```

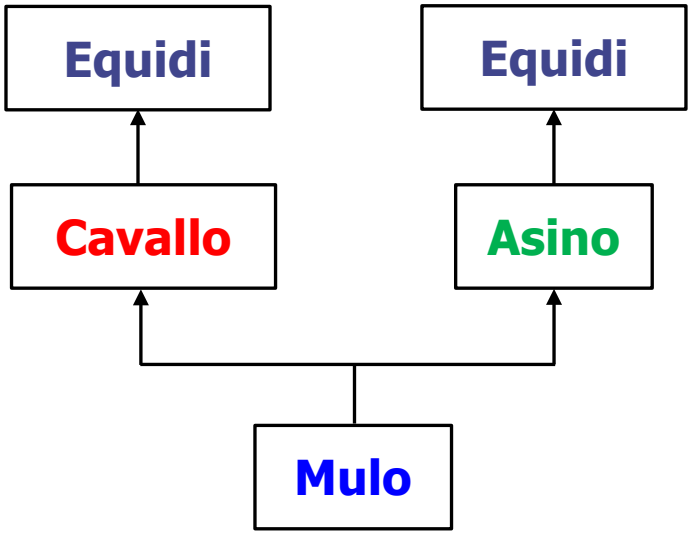
In questo modo grazie alla derivazione multipla la classe **Mulo** erediterà in un sol colpo **tutte le proprietà e tutti i metodi** delle classi **Cavallo** e **Asino**.

N.B. Ovviamente nella lista d'inizializzazione una stessa classe NON PUO' APPARIRE DUE VOLTE COME CLASSE BASE

Linguaggio C++: Ereditarietà (multipla)

A volte si potrebbe anche verificare la situazione, non voluta, che le due classi base derivino, a loro volta, da una stessa classe .

Esempio: Le classi base **Cavallo** ed **Asino** della classe derivata **Mulo**, derivano entrambe dalla classe **Equidi**



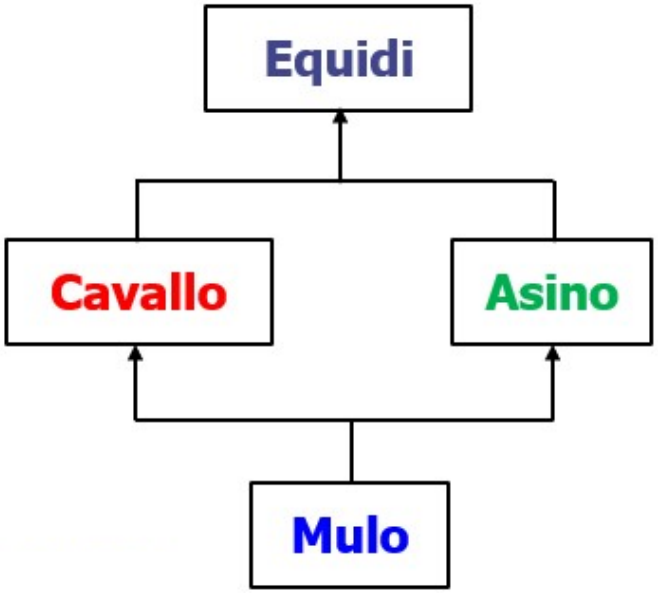
N.B. In questo caso fra i membri (proprietà e metodi) delle classi **Cavallo** ed **Asino** sono presenti anche quelli della classe **Equidi**, che sono così ereditati DUE VOLTE dalla classe **Mulo**.

Linguaggio C++: Ereditarietà (multipla)

Nel linguaggio C++ in questo esempio l'implementazione dell'**ereditarietà multipla con RIPETIZIONE** dei metodi e delle proprietà della classe **Equidi** è la seguente:

Class **Equidi** {...};
Class **Cavallo** : public **Equidi** {...};
Class **Asino**: public **Equidi** {...};
Class **Mulo**: public **Cavallo**, public **Asino** {...};

Poiché non è necessario che i metodi e proprietà della classe **Equidi** compaiano più di una volta nella classe **Mulo**, la struttura gerarchica che desideriamo implementare è quella che ha la seguente forma **a rombo o a diamante**

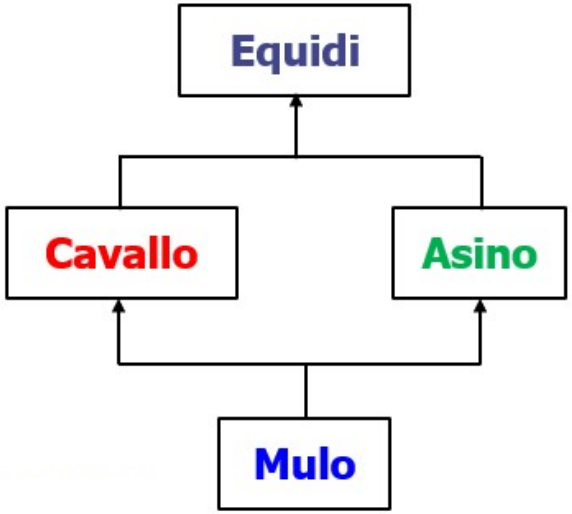


Linguaggio C++: Ereditarietà (multipla)

Questa soluzione ha il pregio di far si che classi "sorelle", in questo caso **Cavallo** ed **Asino**, ereditino le stesse caratteristiche della *parent class* **Equidi** TRASFERENDOLE SOLO UNA VOLTA alla classe **Mulo**.

Il linguaggio C++ consente di implementare questa astrazione mediante la **parola chiave virtual**

Class **Equidi** {...};
Class **Cavallo** : **virtual** public **Equidi** {...};
Class **Asino**: **virtual** public **Equidi** {...};
Class **Mulo**: public **Cavallo**, public **Asino** {...};



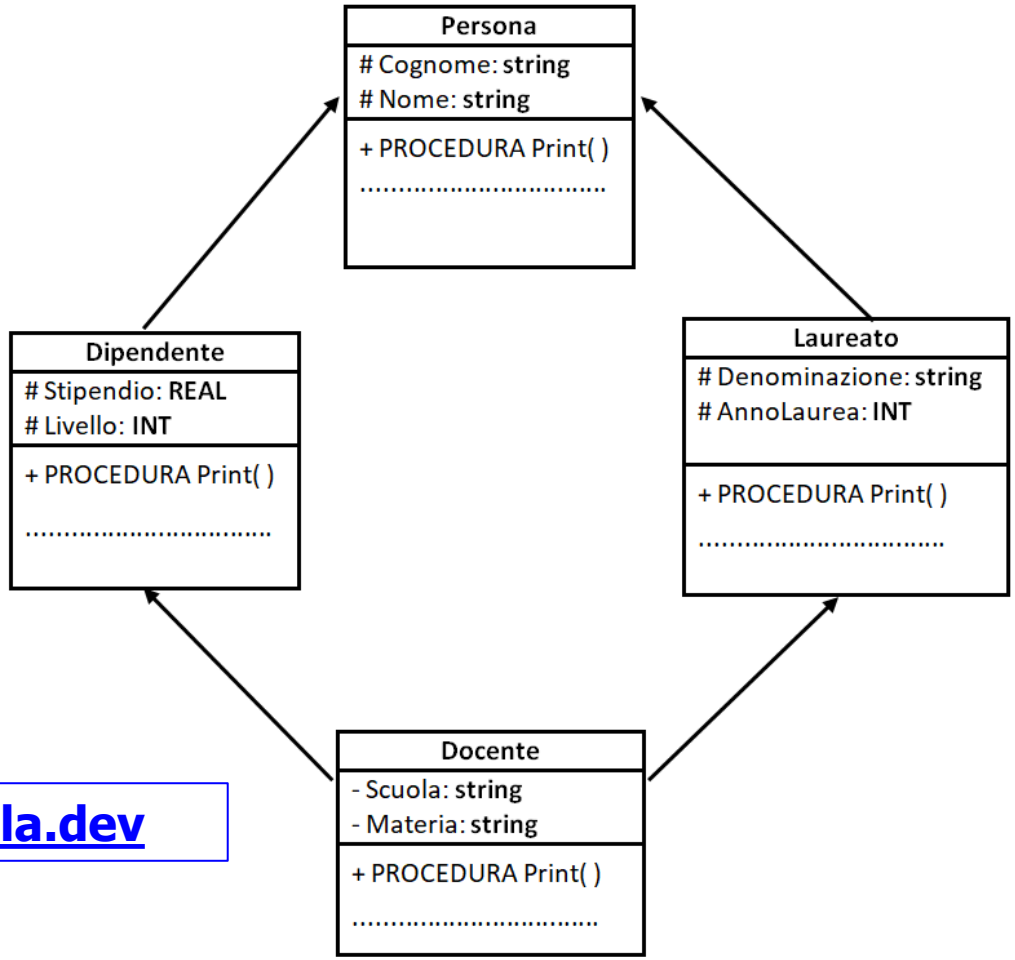
Questa sintassi dichiara la parent class **Equidi** come **virtuale** e fa si che i suoi membri compaiano una sola volta nelle classi ottenute per derivazione successiva

N.B. I meccanismi di ereditarietà semplice ed ereditarietà multipla e di classi virtuali possono essere combinati a piacere per ottenere sistemi raffinati e potenti

Linguaggio C++: Ereditarietà (multipla) - Esercizio

L'implementazione di tale tipo di ereditarietà non è immediata e richiede un'ottima conoscenza del linguaggio C++ e del principio di OOP illustrato.

In uno dei file di esercizi svolti allegati a queste slide, è stato sviluppato un esempio di ereditarietà multipla in un contesto il cui diagramma UML semplificato è il seguente:



15. vedi Ereditarieta_Multipla.dev

Linguaggio C++: Esercizi da svolgere

Esercizio 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** ( ) : **Rettangolo** + **DISTRUTTORE** **~Rettangolo** ( )

18. vedi Rettangolo_1.dev

Linguaggio C++: Esercizi da svolgere

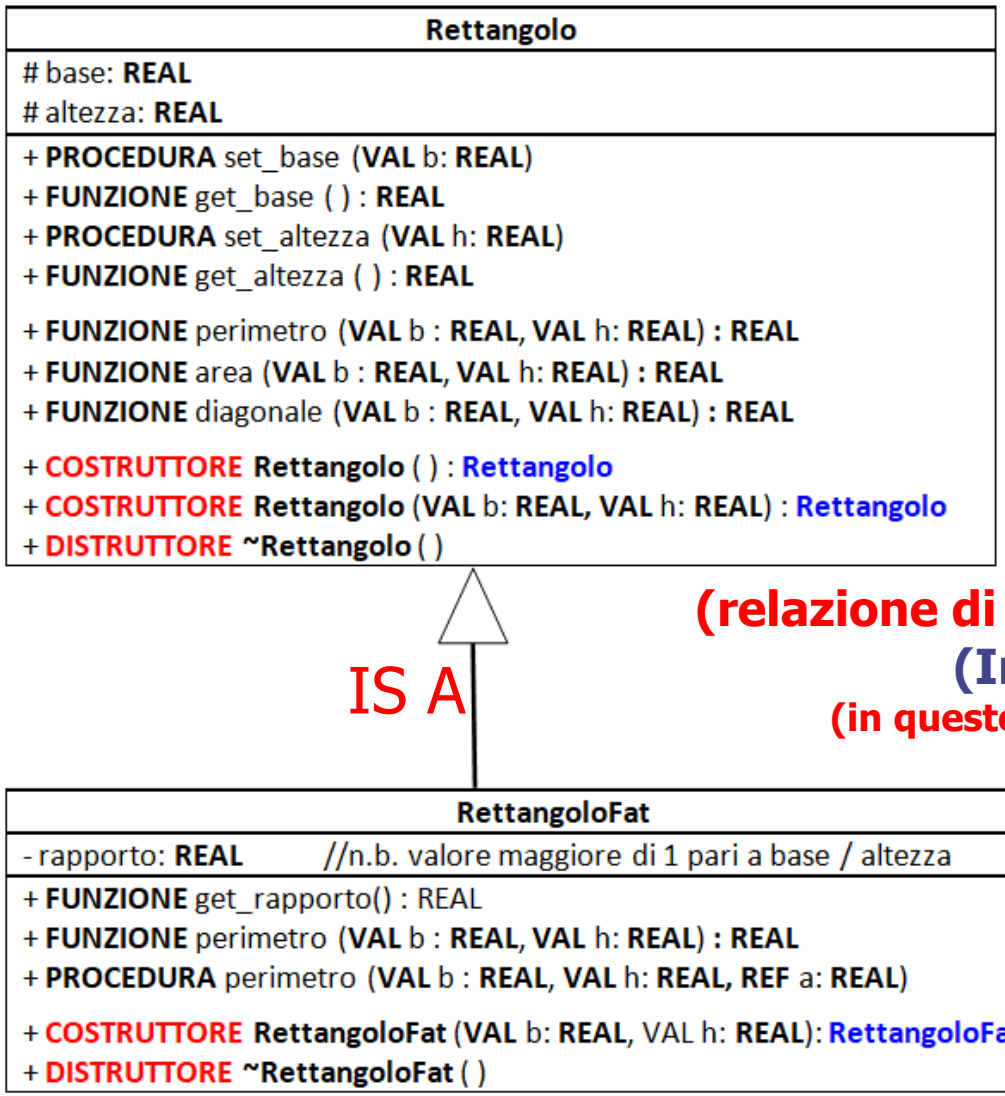
Esercizio classe **Rettangolo** con proprietà **private** e funzioni membro **public**

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

19. vedi Rettangolo_2.dev

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** ( ) : **Rettangolo** + **COSTRUTTORE** **Rettangolo** (**VAL** b: **REAL**, **VAL** h: **REAL**) : **Rettangolo** + **DISTRUTTORE** **~Rettangolo** ( )

Linguaggio C++: Esercizi da svolgere



Esercizio: 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)

20. vedi Rettangolo_3.dev

Linguaggio C++: Esercizi da svolgere

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

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

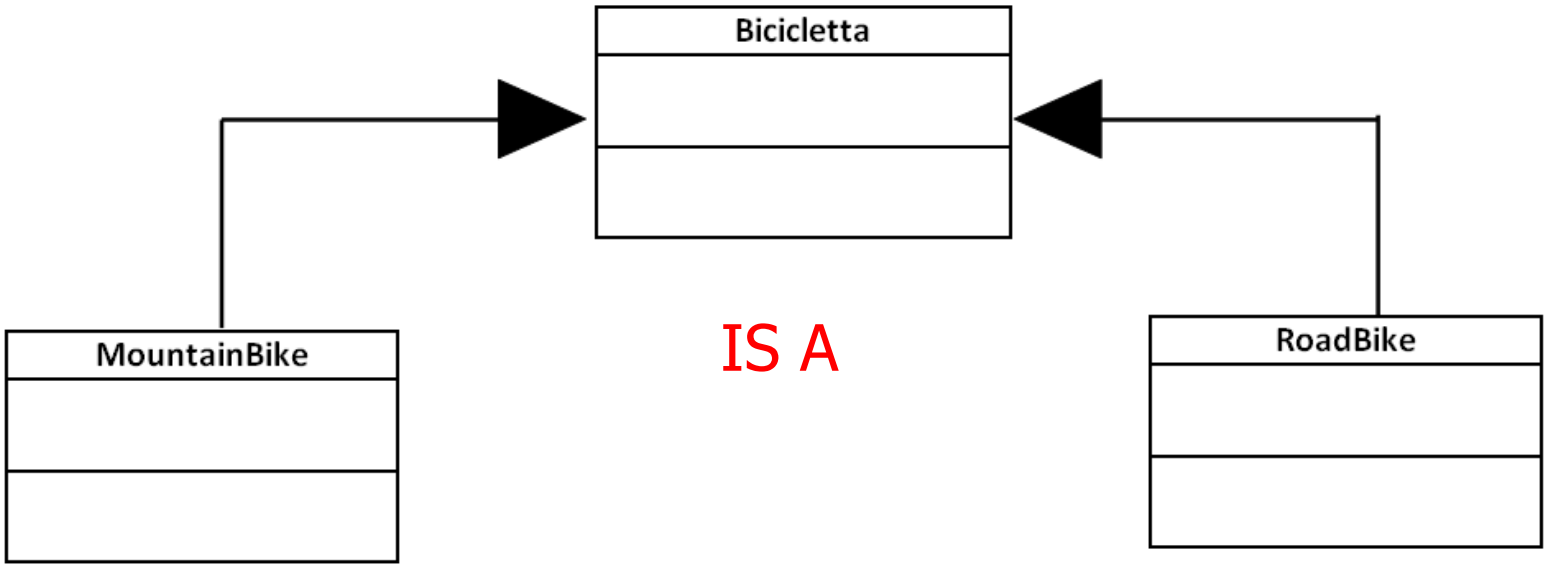
Bicicletta
- cadenza: **INT** - marcia: **INT** - velocita: **INT**
+ **PROCEDURA** set_cadenza (**VAL** c: **INT**) + **FUNZIONE** get_cadenza ( ) : **INT** + **PROCEDURA** set_marcia (**VAL** m: **INT**) + **FUNZIONE** get_marcia ( ) : **INT** + **PROCEDURA** set_velocita (**VAL** v: **INT**) + **FUNZIONE**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

21. vedi Bicicletta_1.dev

Linguaggio C++: Esercizi da svolgere

Esercizio 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)

22. vedi Bicicletta_2.dev

Linguaggio C++: Esercizi da svolgere

Esercizio (continua) **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** ( ) : **Bicicletta** + **COSTRUTTORE** **Bicicletta** (**VAL** c: **INT**, **VAL** m: **INT**, **VAL** v: **INT**) : **Bicicletta** + **DISTRUTTORE** **~Bicicletta** ( )

22. vedi Bicicletta_2.dev

Linguaggio C++: Esercizi da svolgere

Esercizio (continua) **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** accelera (**VAL** inc : **INT, VAL** attr: **INT**) + **PROCEDURA** mostra_stato ( ) + **COSTRUTTORE** **MountainBike** (**VAL** c: **INT, VAL** v: **INT, VAL** m: **INT, VAL** n: **INT, VAL** t: **STRING**) : **MountainBike** + **DISTRUTTORE** **~MountainBike** ( )

22. vedi Bicicletta_2.dev

Linguaggio C++: Esercizi da svolgere

Esercizio (continua) **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** v: **INT**, **VAL** m: **INT**, **VAL** p: **STRING**, **VAL** t: **STRING**) : **RoadBike** + **DISTRUTTORE** **~RoadBike** ( )

22. vedi Bicicletta_2.dev

Linguaggio C++: Esercizi da svolgere

Esercizio 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 () : 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)

23. vedi Crono_1.dev: progetto monofile

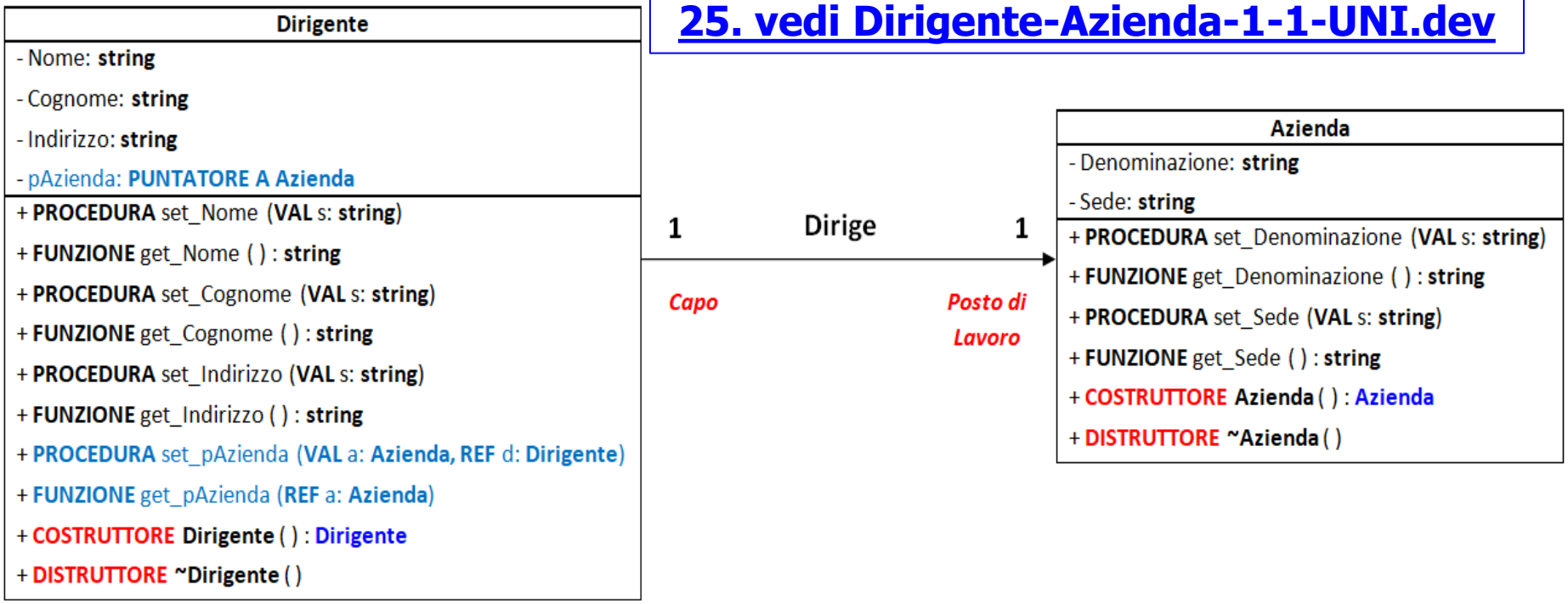
24. vedi Crono_2.dev: progetto multifile

Linguaggio C++: Esercizi da svolgere

Esercizio Dirigente-Azienda molteplicità 1:1 UNIDIREZIONALE

Ecco la relazione binaria di **associazione (Use Relationship) unidirezionale di molteplicità 1:1** tra le classi **Dirigente** ed **Azienda** descritta attraverso **il seguente modello delle classi UML**

25. vedi Dirigente-Azienda-1-1-UNI.dev



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

Autore: Prof. Rio Chierego

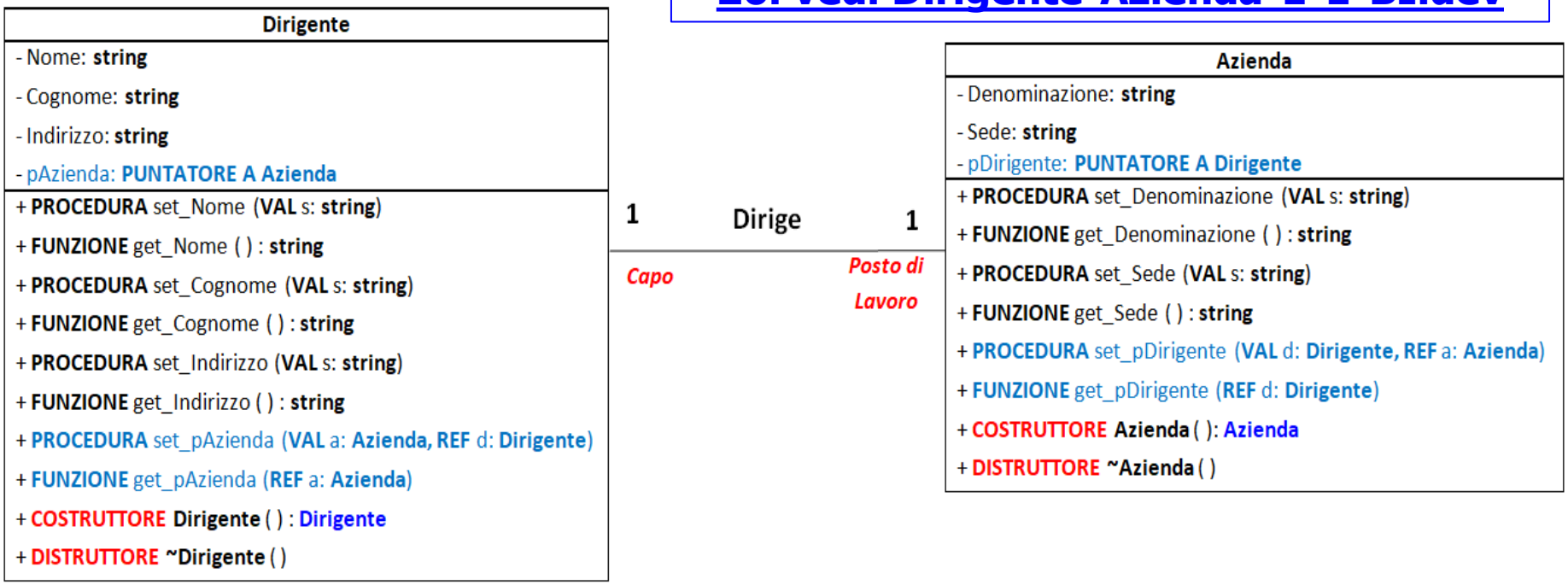
Aprile 2025

Linguaggio C++: Esercizi da svolgere

Esercizio Dirigente-Azienda molteplicità 1:1 BIDIREZIONALE

Ecco la relazione binaria di **associazione (Use Relationship) bidirezionale di molteplicità 1:1** tra le classi **Dirigente** ed **Azienda** descritta attraverso **il seguente modello delle classi UML**

26. vedi Dirigente-Azienda-1-1-BI.dev



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

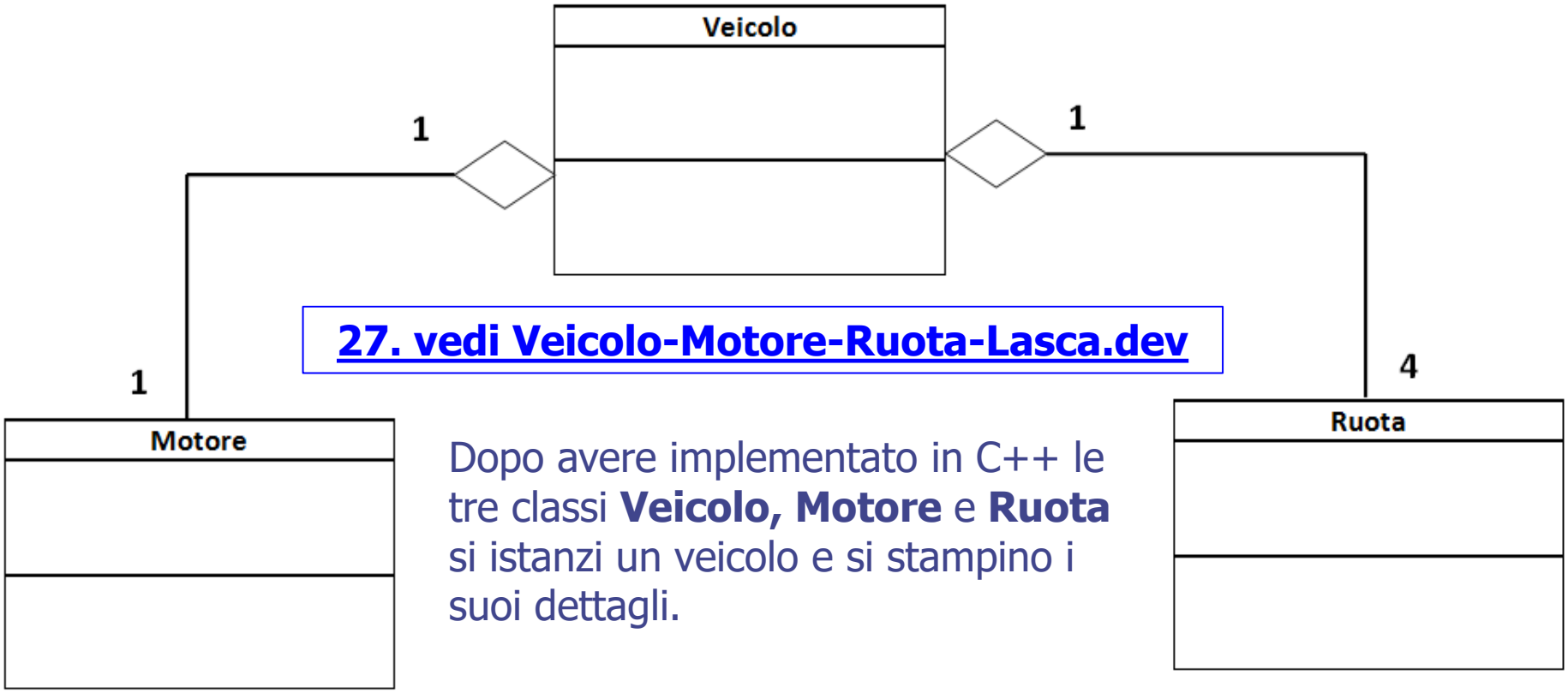
Autore: Prof. Rio Chierego

Aprile 2025

Linguaggio C++: Esercizi da svolgere

Esercizio 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**



27. vedi Veicolo-Motore-Ruota-Lasca.dev

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 da svolgere

Esercizio (continua) **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

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**) : **Veicolo** + **DISTRUTTORE ~Veicolo** ( )

27. vedi Veicolo-Motore-Ruota-Lasca.dev

Linguaggio C++: Esercizi da svolgere

Esercizio (continua) **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** ( ) : **Motore**
+ **DISTRUTTORE** **~Motore** ( )

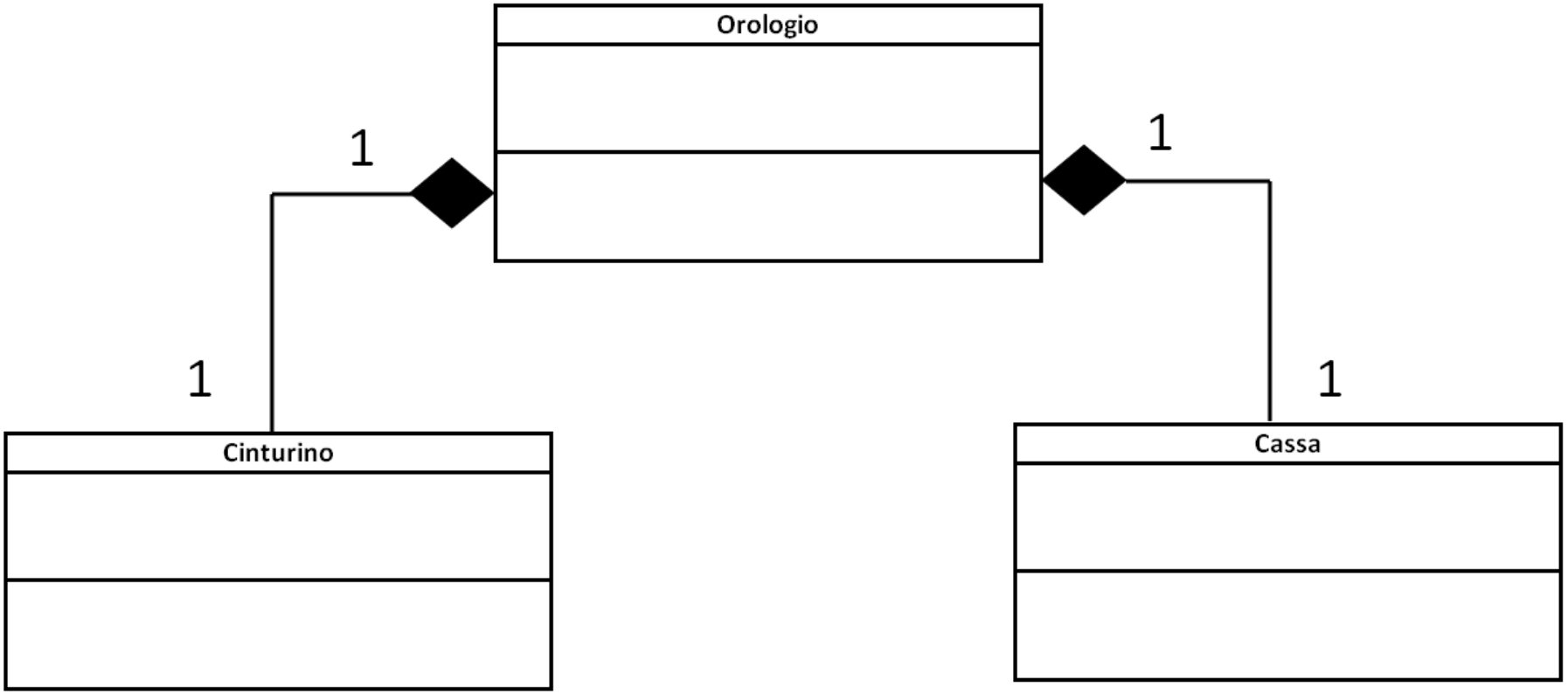
Ruota
- Diametro: **REAL**
- Battistrada: **REAL**
+ **PROCEDURA** set_Diametro (**VAL** d: **REAL**)
+ **FUNZIONE** get_Diametro ( ) : **REAL**
+ **PROCEDURA** set_Battistrada (**VAL** b: **REAL**)
+ **FUNZIONE** get_Battistrada ( ) : **REAL**
+ **PROCEDURA** datiRuota ()
+ **COSTRUTTORE** **Ruota**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

27. vedi Veicolo-Motore-Ruota-Lasca.dev

Linguaggio C++: Esercizi da svolgere

Esercizio Aggregazione STRETTA Orologio-Cassa-Cinturino

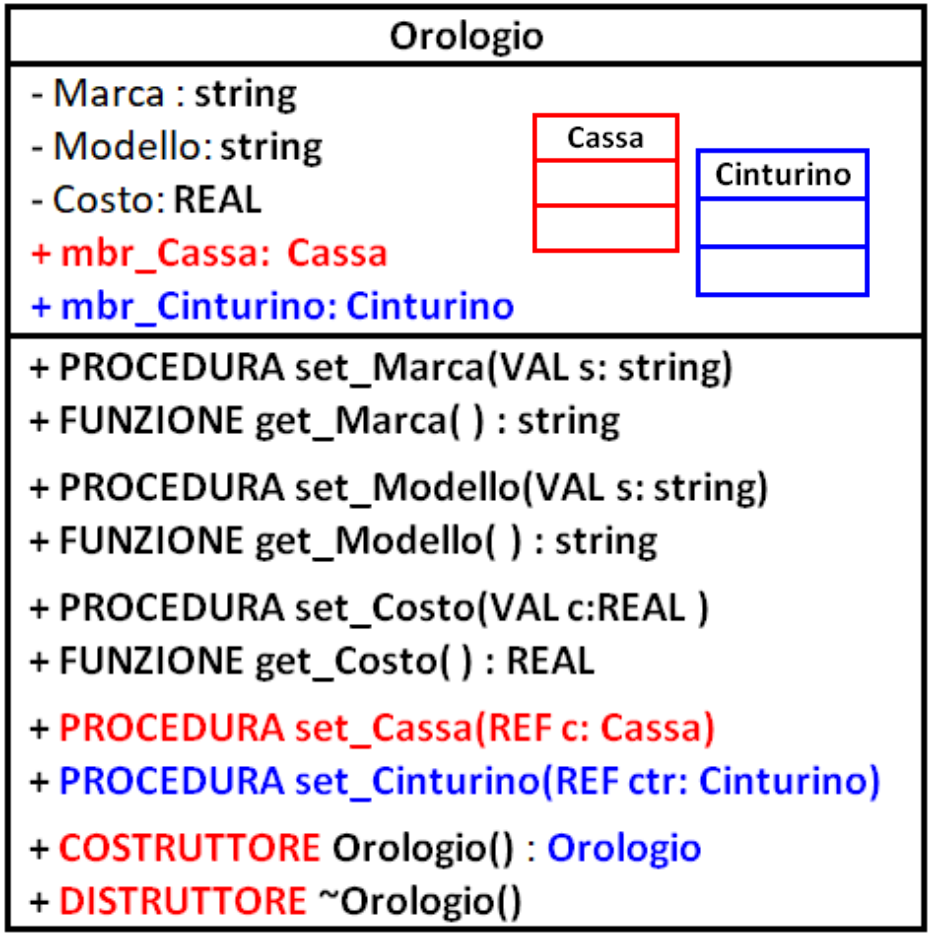
Ecco la relazione di **aggregazione (Containment Relationship) stretta (o forte)** tra le classi **Orologio** (**classe contenitore**) e le classi **Cassa** e **Cinturino** (**classi componenti**) descritta attraverso il seguente modello delle classi UML



28. vedi Orologio-Cassa-Cinturino-Stretta.dev

Linguaggio C++: Esercizi da svolgere

Esercizio (continua) **Aggregazione STRETTA Orologio-Cassa-Cinturino**



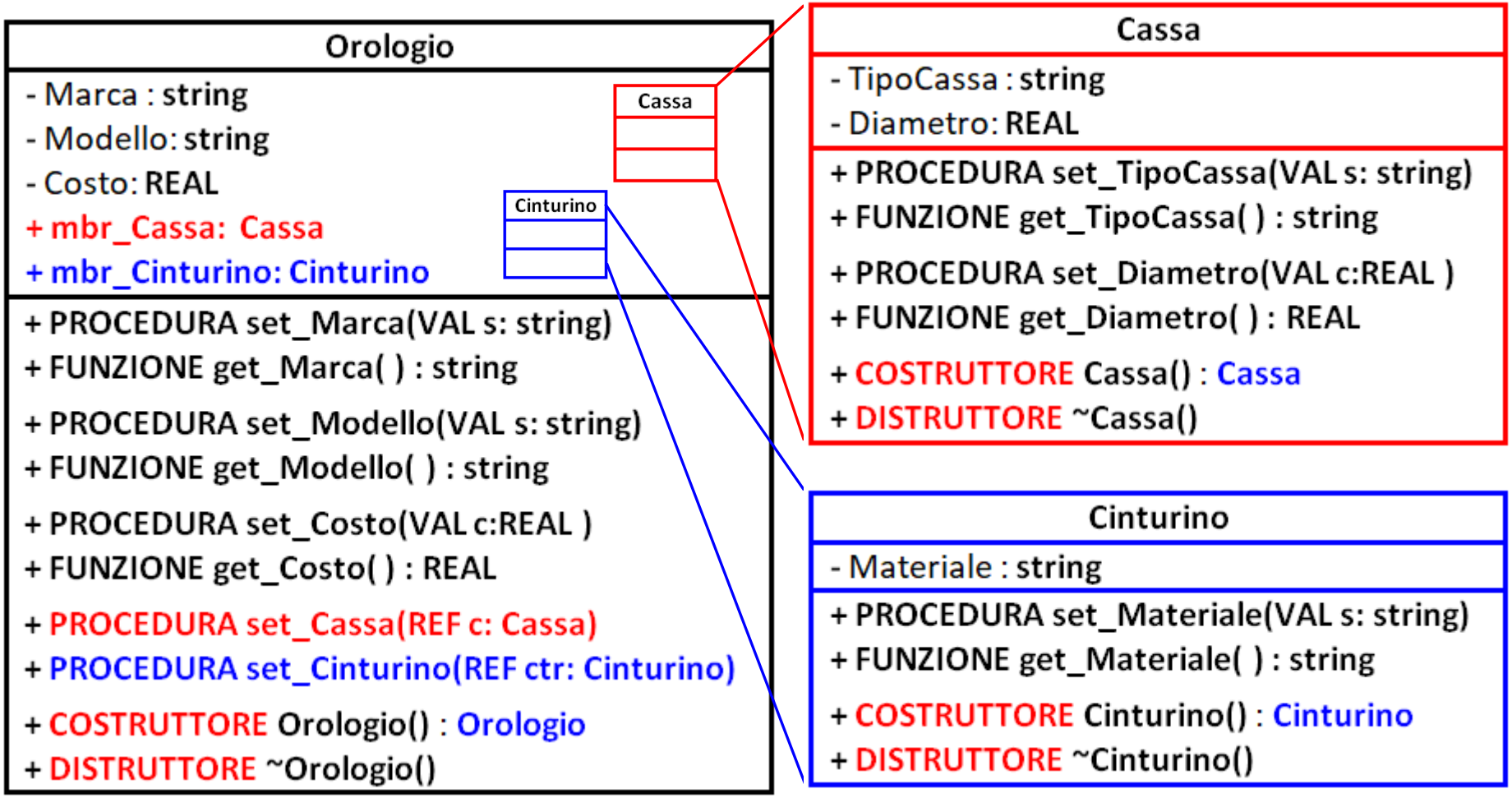
Tale relazione si realizza in C++ **definendo le classi/gli oggetti contenute/i** (Cassa e Cinturino) **direttamente all'interno della classe contenitore** (Orologio) tramite il meccanismo delle **nested classes**

Conseguenza: Quando verrà costruita/distrutta la classe contenitore verranno automaticamente costruite/distrutte anche le classi/gli oggetti contenute/i.

28. vedi Orologio-Cassa-Cinturino-Stretta.dev

Linguaggio C++: Esercizi da svolgere

Esercizio (continua) **Aggregazione STRETTA Orologio-Cassa-Cinturino**



28. vedi Orologio-Cassa-Cinturino-Stretta.dev

Linguaggio C++: Esercizi EXTRA

Tutti gli esercizi EXTRA sono presenti nel file zippato
CHIEREGO-OOP-Programming-Linguaggio-CPP-EXTRA.zip

EXTRA n. 1: Classe Contatore

Realizza una classe **Contatore** che contiene una variabile intera 'Valore' come attributo ed i seguenti metodi:
- un metodo **set_Valore** che setta 'Valore ' con un valore intero deciso dall'utente.
- un metodo **get_Valore** che mostra semplicemente 'valore'.
- un metodo **Plus** che incrementa semplicemente 'Valore ' di 1 tutte le volte che viene invocato.
- un metodo **Minus** che decrementa semplicemente 'Valore ' di 1 tutte le volte che viene invocato.
- un metodo costruttore senza parametri che inizializza 'Valore' a 0

Esegui il test del programma impostando il valore v iniziale di 'Valore' ed il valore n (positivo, nullo o negativo) delle volte che intendi cambiare 'Valore'

N.B. Si rendano esplicitino le scelte implementative fatte nei commenti all'inizio del codice.
Si rispettino i dettami della programmazione orientata agli oggetti con gli attributi privati e i metodi pubblici

Linguaggio C++: Esercizi EXTRA

EXTRA n. 2: Classe Cesare (1/2)

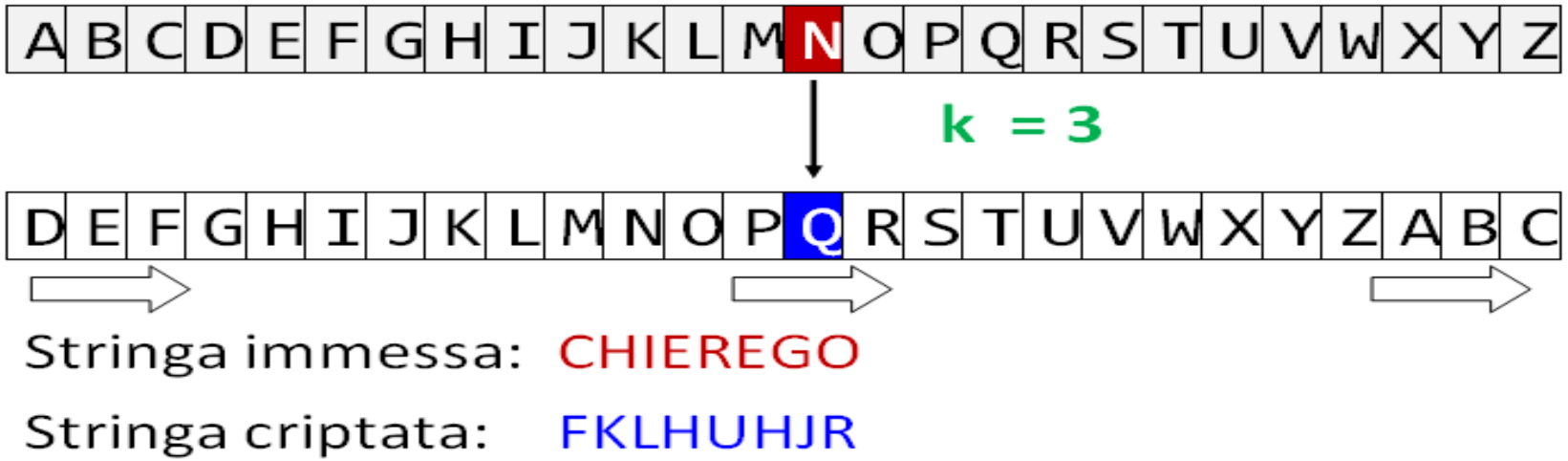
Implementare un algoritmo di crittografia basato su una classe **Cesare** che codifica una stringa di caratteri in ingresso (caratteri ammessi dalla 'A' alla 'Z') in una stringa criptata e resa illeggibile secondo una chiave numerica **k** (con k numero intero qualsiasi).
Il sistema, attraverso questa **chiave numerica k**, prenderà tutte le occorrenze di ogni lettera presente nella stringa di caratteri in ingresso e le sostituirà con la lettera che si trova, seguendo l'ordine alfabetico,
- **k posizioni più a destra:** se k è positivo,
- **k posizioni più a sinistra:** se k è negativo.

In aggiunta ai metodi setter e getter relativi alla stringa di caratteri in ingresso ed alla chiave numerica k, occorrerà implementare anche il metodo getter relativo alla stringa crittografata.

Il metodo pubblico **Conversione()** invece si occuperà di trasformare la stringa di caratteri in ingresso nella stringa criptata ed impostare la relativa proprietà della classe.

Linguaggio C++: Esercizi EXTRA

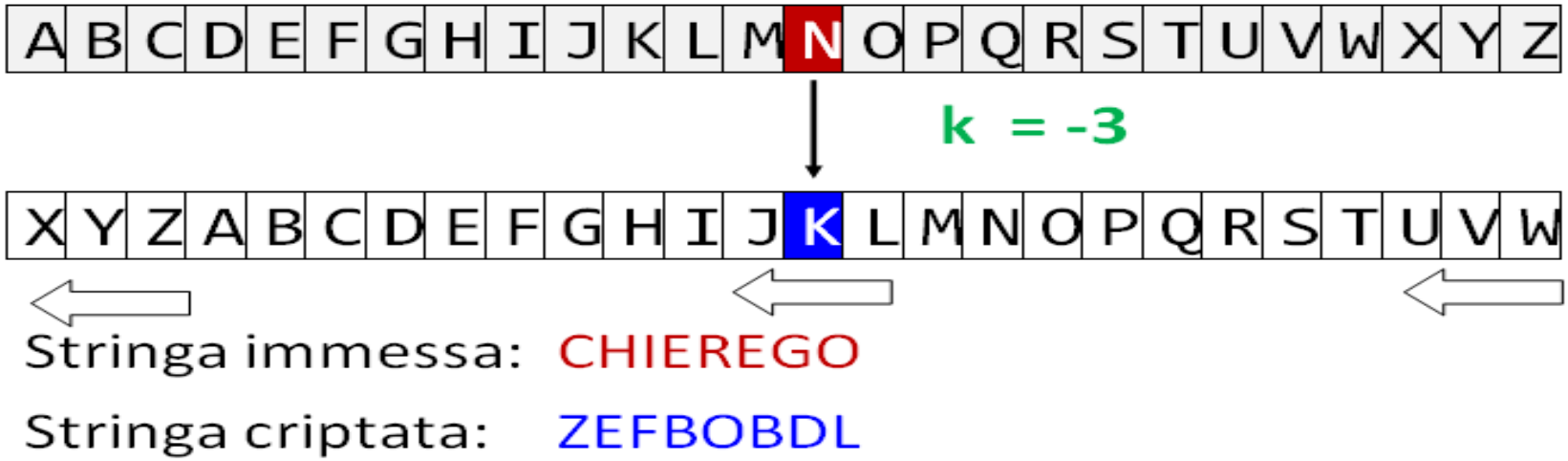
Esempio CHIAVE POSITIVA: k = 3



Stringa immessa: CHIEREGO

Stringa criptata: FKLHUHJR

Esempio CHIAVE NEGATIVA: k = -3



Stringa immessa: CHIEREGO

Stringa criptata: ZEFBOBDL

Linguaggio C++: Esercizi EXTRA

EXTRA n. 3: Classe Televisore

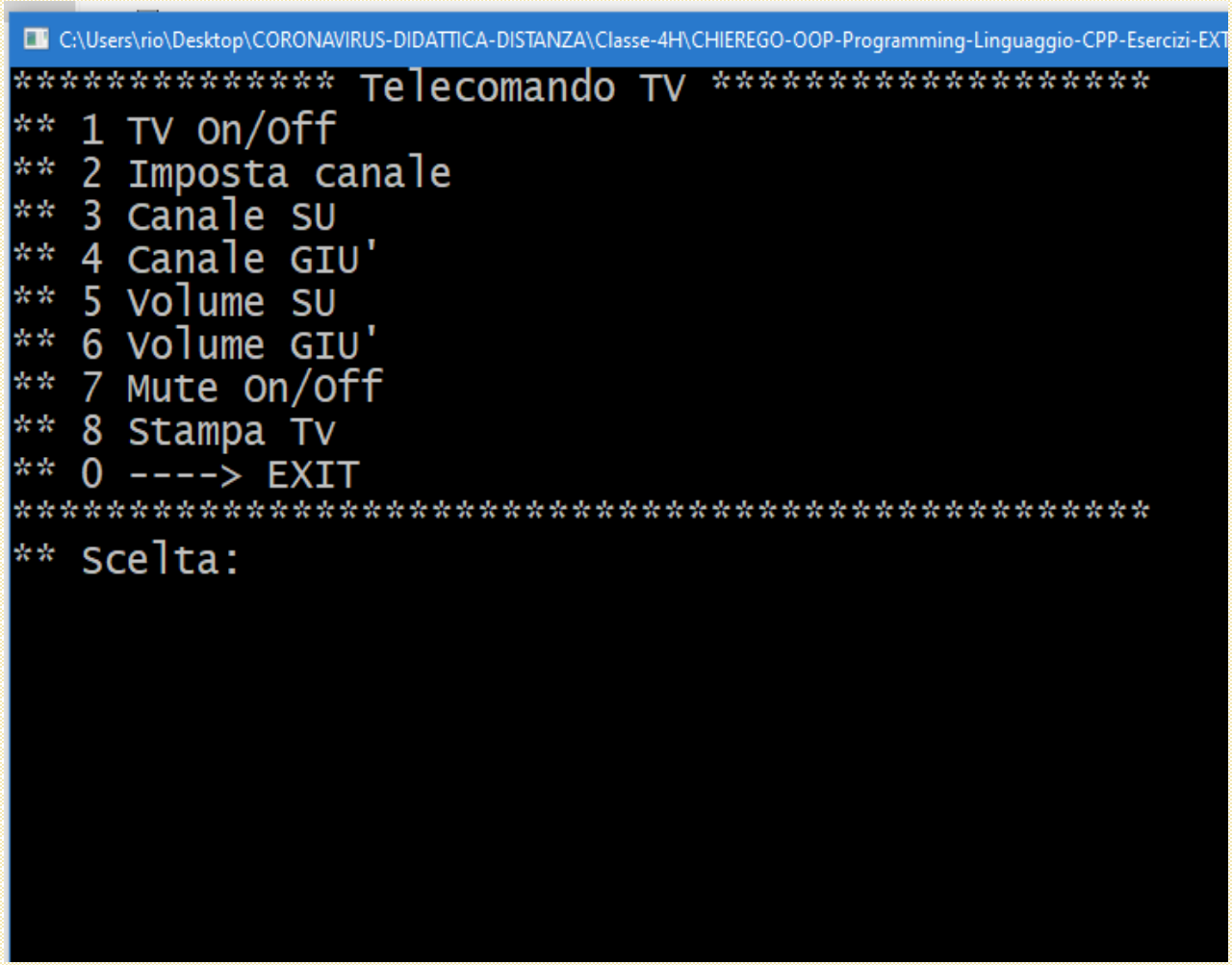
Creare una classe **Televisore** sapendo che lo stato di un televisore è caratterizzato dal fatto di essere acceso o spento, dal volume (che è compreso tra zero e 10), dal canale (che è compreso tra 0 e 99) e dal fatto che sia in modalità silenzioso o meno.

Creare un opportuno costruttore e i seguenti metodi:
- **PulsanteAccensione()**:ad ogni attivazione, setta ed unsetta il valore del pulsante di accensione
- **ImpostaCanale()**: se la tv è accesa, setta un numero di canale prestabilito passato in input
- **CanaleSuccessivo()**:se la tv è accesa, passa al canale successivo, se possibile
- **CanalePrecedente()**:se la tv è accesa, passa al canale precedente, se possibile
- **AumentaVolume**():se la tv è accesa, aumenta di 1 il volume, se possibile
- **AbbassaVolume**():se la tv è accesa, diminuisce di 1 il volume, se possibile
- **PulsanteSilenzioso**():se la tv è accesa, setta ed unsetta il valore del pulsante Mute
-**PrintTv**():se la tv è accesa, visualizza lo stato della tv (il valore di tutte le sue proprietà in quell'istante)

Testare la classe con un opportuno main che preveda un menù di scelta utente (vedi figura) attraverso il quale, attivando tutti i suoi metodi, si simuli un telecomando

Linguaggio C++: Esercizi EXTRA

EXTRA n. 3: Classe Televisore



Linguaggio C++: Esercizi EXTRA

EXTRA n. 3: Classe Televisore (telecomando più user-friendly)

Autore: Prof. Rio Chierego

Aprile 2025

Linguaggio C++: Esercizi EXTRA

EXTRA n. 4: Classe Squadra

Creare una classe **Squadra** che rappresenta una squadra di calcio e ha come attributi il nome, il totale dei punti in classifica, il numero di partite vinte, il numero di partite perse e il numero di partite pareggiate in un intero campionato
(N.B. si controlli che la somma di partite vinte, partite e partite perse sia pari ad un determinato valore costante)

La classe possiede opportuni metodi per impostare le proprietà previste e farle visualizzare, inoltre ha un metodo **CalcolaPunti** per calcolare e valorizzare il totale punti che una squadra ha in campionato (ogni partita vinta vale 3 punti, ogni partita pareggiata 1 punto, ogni partita persa 0 punti), un metodo **StampaSquadra** per visualizzare tutte le proprietà impostate ed un metodo costruttore che imposta il nome della squadra inizializzando a zero il totale punti, il numero di partite vinte, pareggiate e perse.

Creare un main per provare la classe creando due istanze **Napoli** e **Juventus** e si provino ad utilizzare facendo inserire all'utente per entrambe le squadre il numero di partite vinte, perse e pareggiate per poi confrontare quale delle due ha ottenuto più punti in campionato
(N.B. Chiunque assegni nel proprio main di prova più punti alla Juventus, sarà bocciato a giugno!)

N.B.: Si rendano esplicitino le scelte implementative fatte nei commenti all'inizio del codice.
Si rispettino i dettami della programmazione orientata agli oggetti con gli attributi privati e i metodi pubblici

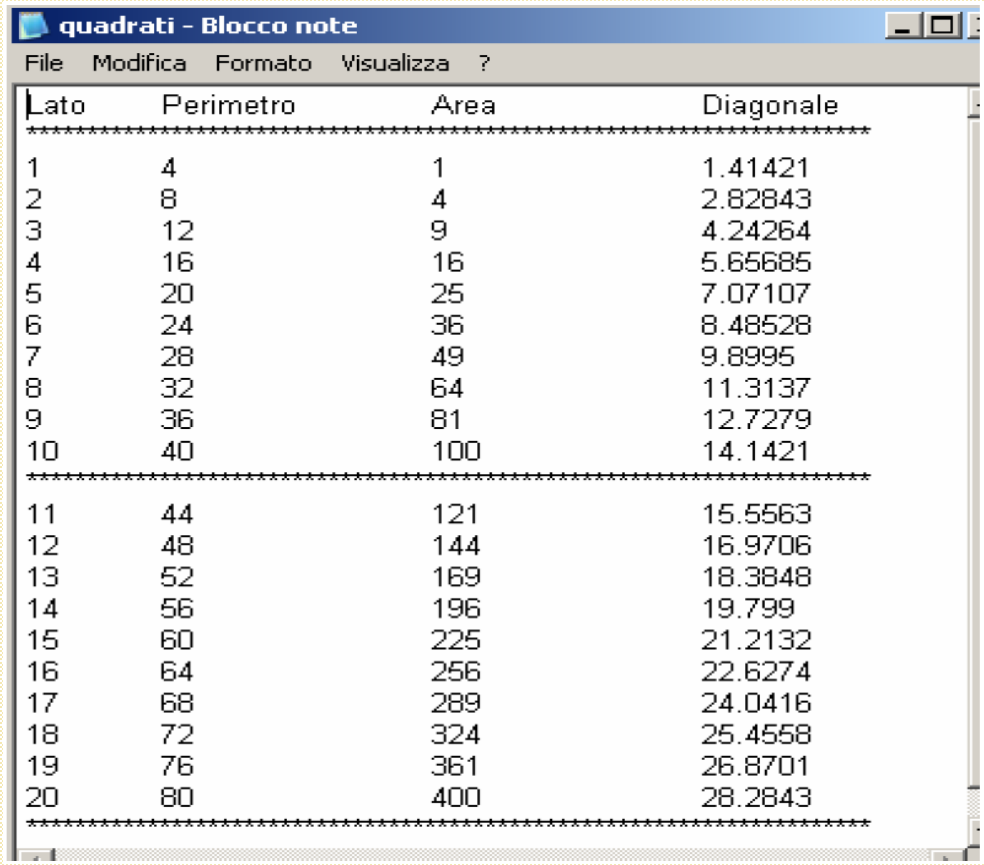
Autore: Prof. Rio Chierego

Aprile 2025

Linguaggio C++: Esercizi EXTRA

EXTRA n. 5: Classe Quadrato

Utilizzando il linguaggio C++ definire una classe **Quadrato** con metodi che calcolino i suoi elementi geometrici fondamentali (perimetro, area e diagonale) più un metodo aggiuntivo CreaFile che permetta di memorizzare nel file sequenziale di testo Quadrati.txt, perimetro, area e diagonale dei quadrati di lato uguale ad 1,2,3…..n-1, n (con n compreso tra 20 e 100)



Lato	Perimetro	Area	Diagonale
1	4	1	1.41421
2	8	4	2.82843
3	12	9	4.24264
4	16	16	5.65685
5	20	25	7.07107
6	24	36	8.48528
7	28	49	9.8995
8	32	64	11.3137
9	36	81	12.7279
10	40	100	14.1421
11	44	121	15.5563
12	48	144	16.9706
13	52	169	18.3848
14	56	196	19.799
15	60	225	21.2132
16	64	256	22.6274
17	68	289	24.0416
18	72	324	25.4558
19	76	361	26.8701
20	80	400	28.2843

Specifiche: il file di testo deve presentarsi come nell'esempio, con la intestazione e una riga di 70 asterischi ogni "decade"

Linguaggio C++: Esercizi EXTRA

EXTRA n. 6: Classe Scatola

Creare una classe **Scatola** con gli opportuni attributi e metodi, per ogni scatola bisognerà memorizzare la dimensione dei lati e il suo peso, si potrà poi avere a disposizione le caratteristiche della scatola, ovvero le dimensioni dei lati, l'area della base, il volume, il peso, la densità media.

Creare la classe **GiocoInScatola** in cui andranno memorizzate oltre alle informazioni della Scatola anche il numero di giocatori e il nome del gioco.
Si crei inoltre per entrambe le classi un metodo **Print()** che permette di stampare a schermo i valori degli attributi dell'istanza sul quale è richiamato.
Testare le classi con un opportuno main.
Tra le prove, far inserire le informazioni di due giochi da tastiera e poi dire quali tra i due ha il rapporto volume/numero di giocatori maggiore

N.B.
Si rendano esplicitino le scelte implementative fatte nei commenti all'inizio del codice. Si rispettino i dettami della programmazione orientata agli oggetti con gli attributi privati e i metodi pubblici

Linguaggio C++: Esercizi EXTRA

EXTRA n. 7: Classe Bicicletta (1/2)

Creare una classe **Bicicletta** che memorizzi per ogni bicicletta: **taglia** del telaio (XS, S, M, L, XL, XXL), **marca** e la **velocità** a cui sta andando.

Creare tutti i **metodi** opportuni della classe Bicicletta, inoltre si creino:
– **void StampaStringa()** che restituisca una stringa con le caratteristiche della bicicletta

ad esempio
"Bicicletta: **Marca**: Atala - **Telaio**: XL - **Velocità**: 10 km/h"

– void StampaStato () che stampa a schermo lo stato

ad esempio "**Velocità**: 10 km/h"

Linguaggio C++: Esercizi EXTRA

EXTRA n. 7: Classe Bicicletta (2/2)

Creare una classe **MountainBike**: ogni mountain bike è una Bicicletta che ha la caratteristica specifica di possedere **l'ammortizzazione** (a molla oppure ad olio) e le **marce** gestite da due selettori.

I valori delle marce andranno:
- per il primo selettore da 1 a un massimo di 3
- per il secondo selettore da 11 ad un massimo di 60.

Creare tutti i metodi opportuni della classe MountainBike, inoltre si creino:
– void **StampaStringa()** che restituisca una stringa con le caratteristiche della mountain bike

ad esempio
"MountainBike: **Marca**: MERIDA - **Telaio**: XS – **Velocità**: 15 km/h – **Ammortizzata**: a molla – **Marce**: 2+25"

– void **StampaStato()** che stampa a schermo lo stato della mountain bike ovvero ad esempio:
" **Ammortizzata**: a molla – **Marce**: 2+25"

Linguaggio C++: Esercizi EXTRA

EXTRA n. 8: Classe Studente (1/2)

Implementare una classe **Studente** che contenga come **proprietà private** sia le informazioni anagrafiche (Cognome e **Nome**), sia le informazoini relative alla **classe** ed alla **sezione** frequentata.

Inoltre dovranno essere memorizzate anche le informazioni relative ai **voti** (numero intero da 1 a 10) riportati relativi a 10 materie ed alla **media** conseguente.

Tale classe deve contenere, oltre ai metodi **setter** e getter che si riterrà necessari, i seguenti metodi:

- il metodo **Print**() che visualizza a video tutti i valori posseduti da un oggetto istanziato della classe;

- il **costruttore di default** che inizializzerà opportunamente le proprietà dell'oggetto della classe con opportuni valori di default;

- un **costruttore aggiuntivo** che permetta di inizializzare le proprietà dell'oggetto della classe con specifici valori ricevuti dall'esterno;

- il **distruttore**.

Linguaggio C++: Esercizi EXTRA

EXTRA n. 8: Classe Studente (2/2)

Esegui il test del programma costruendo un main nel quale:

- dichiarare un oggetto **s1** della classe **Studente allocato staticamente** con il costruttore di default;

- valorizzare con i relativi metodi setter previsti, le sue proprietà immettendole da tastiera;

- visualizzare lo **stato** dell'oggetto **s1** (ossia le proprietà assegnate all'oggetto s1) tramite il metodo Print();

- dichiarare un oggetto **s2** della classe **Studente allocato dinamicamente** con il costruttore "alternativo" istanziandolo con valori preimpostati;

- visualizzare lo **stato** dell'oggetto **s2** (ossia le proprietà assegnate all'oggetto s2) attraverso i metodi getter previsti;

- effettuare il **confronto** tra le classi e le sezioni dei due oggetti della classe Studente segnalando con opportuni messaggi a video se essi frequentano lo stesso anno di studi ed eventualmente anche la stessa sezione

- effettuare anche il confronto tra le medie voto possedute dai due studenti

Linguaggio C++: Esercizi EXTRA

EXTRA n. 9: Torneo di calcio 1/2 (Università agli Studi "PARTHENOPE")

INGEGNERIA INFORMATICA, BIOMEDICA E DELLE TELECOMUNICAZIONI

PROGRAMMAZIONE DEI CALCOLATORI ELETTRONICI

Proff Roberto Nardone, Luigi Romano

Demo prova intracorso

XX marzo 2022 – 1.5 ore

Si sviluppi un programma C++ che implementi le seguenti specifiche:

1. Si definisca una classe *Squadra* che abbia due attributi privati, *nome* e *annoDiFondazione*. Fornire per la classe gli opportuni costruttori (senza e con due parametri) e l'eventuale distruttore. Definire inoltre i metodi di get e set per ciascuno dei due attributi.
2. Si derivino dalla classe definita al punto precedente, due ulteriori classi *SquadraClub* e *SquadraNazionale*. Definire per la classe *SquadraClub* gli attributi *sedeLegale* e *nazionalità*. Definire per la classe *SquadraNazionale* l'attributo *nazione*. Aggiungere inoltre gli opportuni costruttori e distruttori, ed i soli metodi di get per tutti gli attributi delle classi derivate.
3. Si definisca una classe *Torneo* che contiene una lista di *SquadraClub*. Definire per la classe (oltre a costruttori e distruttori che si ritiene opportuno), il metodo *addSquadra* che prende in ingresso un *SquadraClub* e la aggiunge alla lista delle squadre iscritte al torneo.
4. Si aggiunga alla classe *Torneo* un metodo *removeSquadra* che prende in ingresso un *nome* e rimuove la squadra con quel nome (se esiste).
5. Si aggiunga alla classe *Torneo* un metodo *getSquadre* che prende in ingresso una *nazionalità* e restituisce la lista di squadre con quella nazionalità.

Linguaggio C++: Esercizi EXTRA

EXTRA n. 9: Torneo di calcio 2/2 (Università agli Studi "PARTHENOPE)

Note

È obbligatorio consegnare un programma che compili e vada in esecuzione, contenente un main che chiede le informazioni all'utente tramite terminale e solleciti (tutti) i metodi realizzati.

Il programma può essere strutturato su un unico file oppure su più file (con impatto sulla valutazione). É possibile utilizzare la classe string oppure gestire (non necessariamente tutti) array dinamici di char (char*) tramite le funzioni di manipolazione di stringhe, ovvero strcat, strcpy, ecc. (con impatto sulla valutazione). É possibile gestire gli oggetti con array statici o con array dinamici di oggetti o puntatori (con impatto sulla valutazione).

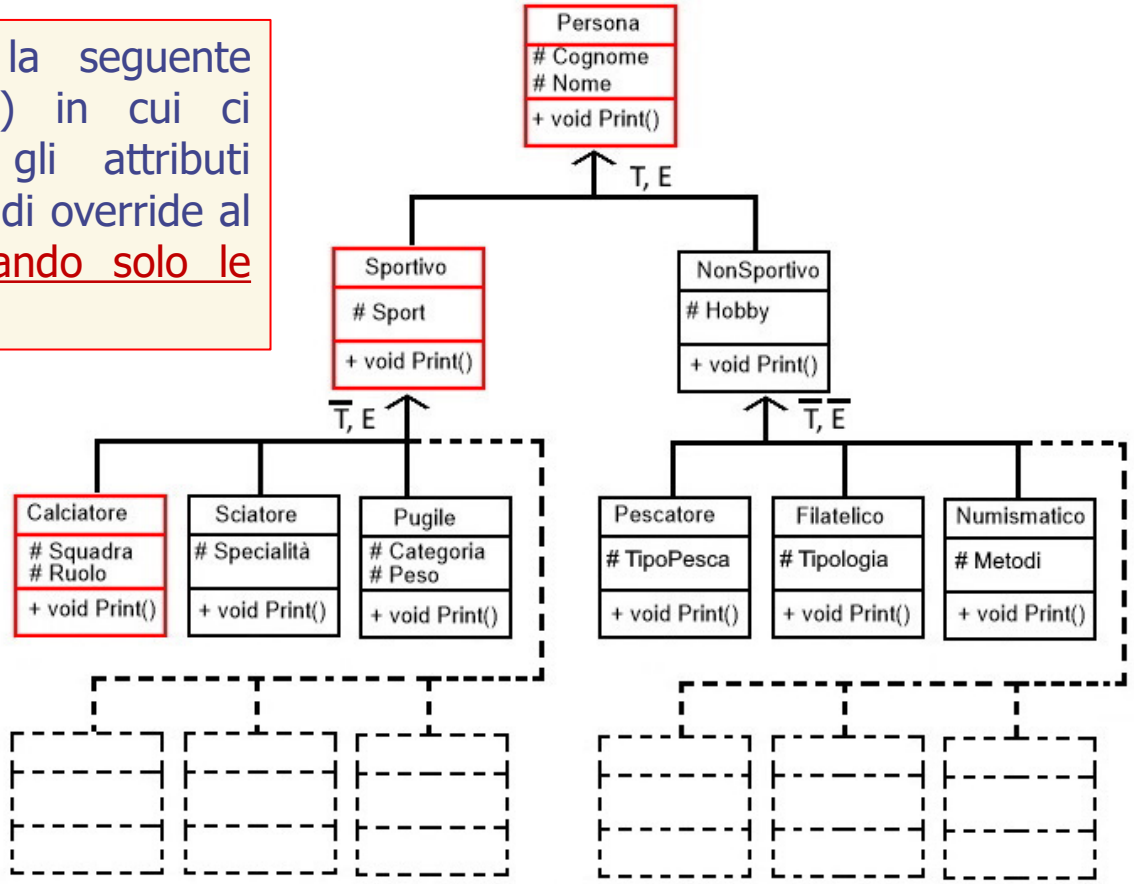
Risoluzione dei punti 1 - 2: 18 (con string e array statici) – 23 (con char* e array di puntatori)
Risoluzione dei punti 1 - 3: 24 (con string e array statici) – 27 (con char* e array di puntatori)
Risoluzione dei punti 1 - 5: 28 (con string e array statici) – 30 (con char* e array di puntatori)

Linguaggio C++: Esercizi EXTRA

EXTRA n. 10: Classe Persona-Sportivo-Calciatore

Si progetti un insieme di classi in grado di descrivere le caratteristiche di alcune persone che possono essere sportivi e non e dove alcuni degli sportivi praticano il gioco del calcio.

E' possibile individuare la seguente gerarchia di classi (ISA) in cui ci limiteremo ad inserire gli attributi essenziali con un esempio di override al **metodo Print()** sviluppando solo le classi bordate di rosso



Linguaggio C++: Esercizi EXTRA

EXTRA n. 11: POKER italiano con 5 DADI

Si progetti la classe **Dado** con la proprietà privata **Faccia** ed il solo metodo **getter** più il metodo **lancia** in grado poter assegnare in modo casuale un valore da 1 a 6.
Si costruisca poi un **main** in grado di gestire il gioco consentendo al giocatore il lancio contemporaneo di 5 dadi valutando opportunamente il punteggio ottenuto applicando il seguente schema:

Pokerissimo : tutti e 5 dadi hanno lo stesso valore	**PUNTI 5**
Poker: 4 dadi su 5 hanno lo stesso valore	**PUNTI 4**
Full: 3 dadi hanno lo stesso valore ed anche altri 2	**PUNTI 4**
Scala: tutti e 5 i dadi hanno valori differenti	**PUNTI 4**
Doppia Coppia: 4 dadi dal valore uguale a 2 a 2	**PUNTI 2**
Coppia:2 dati con lo stresso valore e gli altri 3 no	**PUNTI 1**
In tutti gli altri casi	**PUNTI 0**

Linguaggio C++: Esercizi EXTRA

EXTRA n. 12: La classe **PILA**

Seguendo le specifiche **dell'A.D.T.** (*Abstract Data Type*) viste a lezione, si progetti (e poi si implementi) la classe **Pila** sulla quale sono utilizzabili le sole **operazioni:**

- **Crea ( )**
- **Push ( )**
- **Pop ( )**
- **TestVuota ( )**

Si ricorda che la pila (dall'inglese **STACK**), in informatica, è un tipo di dato astratto che viene usato in diversi contesti per riferirsi a strutture dati, le cui modalità d'accesso agli elementi in essa contenuti seguono la modalità LIFO.

L.I.F.O. Last In First Out = ossia l'ultimo ad entrare è il primo ad uscire!

Esempio concreto di PILA in Informatica: il sistema operativo gestisce a **run-time la PILA** (dei record) **DI ATTIVAZIONE** (o **RDA**):

• per ogni attivazione (o chiamata) ad una funzione viene creato un nuovo RDA ed inserito in cima alla pila (*Push( )*);

• al termine dell'attivazione (o chiamata) della funzione il RDA viene rimosso dalla cima della pila (*Pop( )*)

Linguaggio C++: Esercizi EXTRA

EXTRA n. 12: La classe **PILA** **(implementata con un ARRAY STATICO)**

Suggerimento sul possibile menù di interazione utente con un oggetto della classe **Pila**

```
**********************************************************
*       Menu' utente principale                          *
**********************************************************
* 1 CREA LA PILA                                         *
* 2 PUSH - INSERISCI NODO (IN TESTA ALLA PILA)           *
* 3 POP - PRELEVA NODO (DALLA TESTA DELLA PILA)          *
* 4 TEST VUOTA                                           *
* 5 STAMPA LA PILA                                       *
* 6 STAMPA LA PILA BELLA (beta)                          *
* 0 =======> USCITA                                      *
**********************************************************
```

Sarà poi sempre possibile utilizzare all'interno di un programma un oggetto di tipo Pila sia allocato staticamente, sia allocato dinamicamente

Autore: Prof. Rio Chierego

Aprile 2025

Linguaggio C++: Esercizi EXTRA

EXTRA n. 12: La classe **PILA (implementata con un ARRAY STATICO)**

```
#ifndef _PILA_H_
#define _PILA_H_

#define MAXLEN 10

class Pila
   {
   private:
      //STRUTTURA DATI STATICA, SEQUENZIALE AD ACESSO DIRETTO
      int corpo[MAXLEN];      //Contiene i nodi della PILA
      int top;                //Posizione dell'ultimo nodo presente nella PILA

   public:

      //Operazioni previste in accordo con l'ADT
      int* Crea (void);
      void Push (int testa[], int ele);
      bool Pop (int testa[], int* ele);
      bool TestVuota (int testa[]);

      //Metodi di servizio
      int MostraMenu (void);
      void StampaPila (int testa[]);
      void StampaPilaBella (int testa[]);

   };

#endif
```

I nodi della pila saranno gestiti con una struttura dati **allocata staticamente**, **sequenziale** ed ad **accesso diretto (array)**

Linguaggio C++: Esercizi EXTRA

EXTRA n. 12: La classe **PILA** **(implementata con una LISTA LINKATA DINAMICA)**

Suggerimento sul possibile menù di interazione utente con un oggetto della classe **Pila**

```
**********************************************************
*       Menu' utente principale                          *
**********************************************************
* 1 CREA LA PILA                                         *
* 2 PUSH - INSERISCI NODO (IN TESTA ALLA PILA)           *
* 3 POP - PRELEVA NODO (DALLA TESTA DELLA PILA)          *
* 4 TEST VUOTA                                           *
* 5 STAMPA LA PILA                                       *
* 6 DEALLOCA LA PILA                                     *
* 0 =======> USCITA                                      *
**********************************************************
```

Sarà poi sempre possibile utilizzare all'interno di un programma un oggetto di tipo Pila sia allocato staticamente, sia allocato dinamicamente

Autore: Prof. Rio Chierego

Linguaggio C++: Esercizi EXTRA

EXTRA n. 12: La classe **PILA (implementata con una LISTA LINKATA)**

```
#ifndef _PILA_H_
#define _PILA_H_

#define MAXLEN 10

#include "Nodo.h"

class Pila
   {
   private:
      //STRUTTURA DATI DINAMICA, NON SEQUENZIALE AD ACESSO SEQUENZIALE
      //LISTA SEMPLICEMENTE LINKATA
      NODO* pTesta;     //Contiene il puntatore alla testa della (lista linkata) PILA
      int nodiTot ;     //Numero totali di nodi presenti nella (lista linkata) PILA

   public:

      //Operazioni previste in accordo con l'ADT
      NODO* Crea (void);
      void Push (NODO** pTesta, int ele);
      bool Pop (NODO** pTesta, int* ele);
      bool TestVuota (NODO* pTesta);

      //Metodi di servizio
      int MostraMenu (void);
      void StampaPila (NODO* pTesta);
      void DeallocaPila (NODO* pTesta);

      //Metodi getter e setter della proprietà pTesta
      NODO* get_pTesta(void);
      void set_pTesta(NODO*);

      //Metodo getter e setter della proprietà nodiTot
      int get_nodiTot(void);
      void set_nodiTot(int);

   };
#endif
```

I nodi della pila di tipo **NODO** saranno gestiti con una struttura dati **allocata dinamicamente**, **non sequenziale** ed ad **accesso sequenziale (lista linkata semplice)** attraverso **il puntatore alla testa della lista (pTesta)**

```
#ifndef _NODO_H_
#define _NODO_H_

// definizione della struttura NODO
typedef struct nodo
   {
   int info;
   struct nodo * pNext;
   } NODO;

#endif
```