



Laurea triennale in Informatica

modulo (CFU 6) di

Programmazione II e Lab.

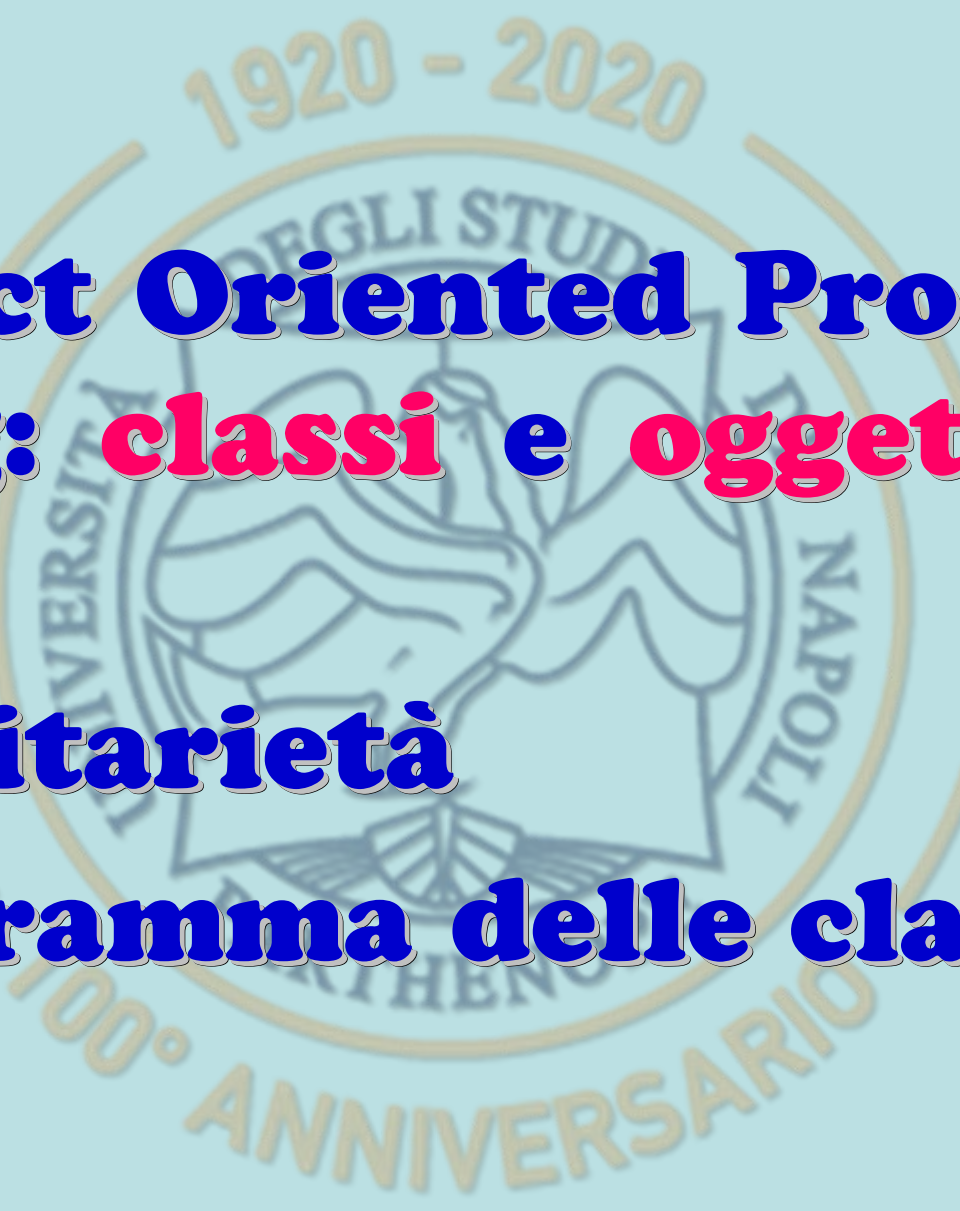
prof. Mariarosaria Rizzardi

Centro Direzionale di Napoli – Isola C4

stanza: n. 423 – IV piano Lato Nord

tel.: 081 547 6545

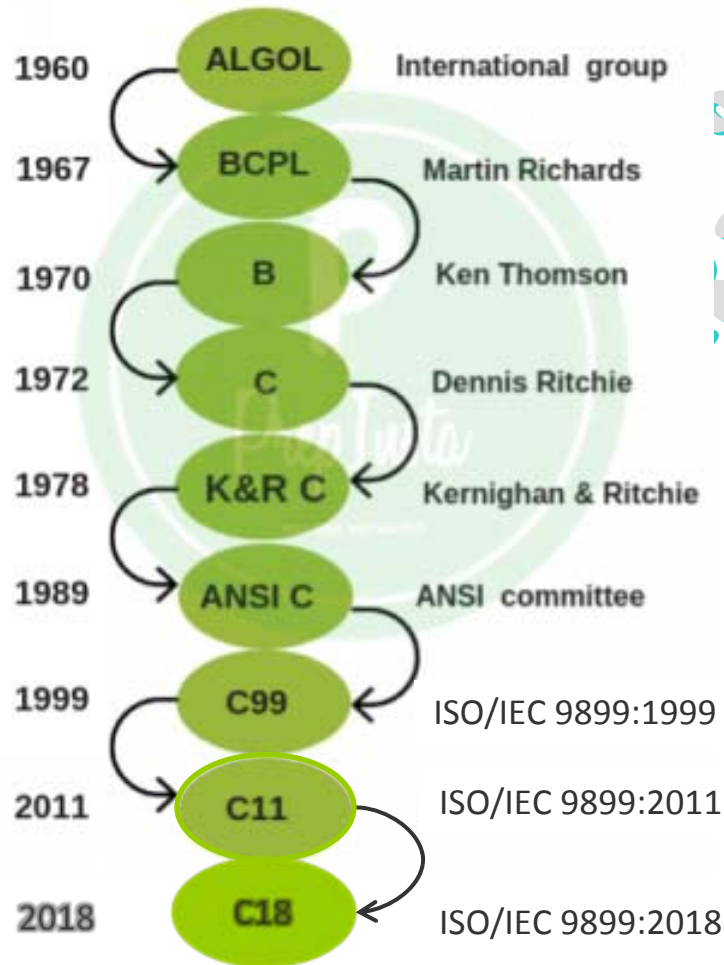
email: mariarosaria.rizzardi@uniparthenope.it

- 
- **Object Oriented Programming: classi e oggetti del C++**
 - **Ereditarietà**
 - **Diagramma delle classi**

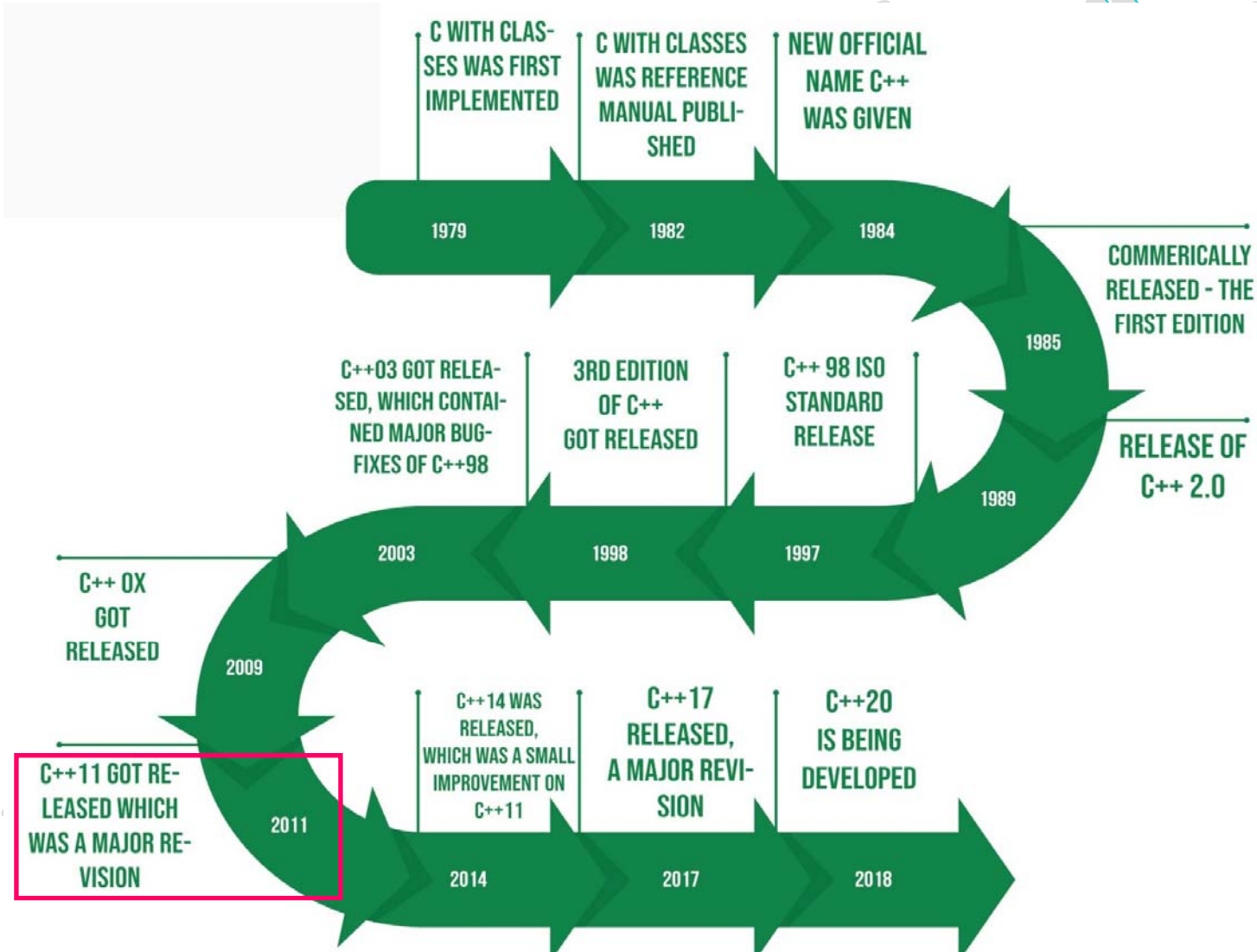
Cronologia dei principali linguaggi di programmazione

Anno	Linguaggio
1958	FORTRAN II (John W. Backus)
1960	ALGOL 60 (Peter Naur, J.W. Backus <i>et al.</i>) COBOL (CODASYL Committee)
1962	FORTRAN IV
1964	BASIC (G.J. Kemeny, T.E. Kurtz)
1966	FORTRAN 66
1968	ALGOL 68 (A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, Cornelis H.A. Koster <i>et al.</i>)
1969	PL/I (IBM)
1970	Pascal (Niklaus Wirth, Kathleen Jensen)
1972	C (Dennis Ritchie)
1978	MATLAB (Cleve Moler) SQL (IBM)
1983	C++ (Bjarne Stroustrup, AT&T)
1991	Python (Guido Van Rossum)
1995	Java (James Gosling, Sun Microsystem)
2001	C# (Anders Heijlsberg, Microsoft)

History of C



History of C++



Alla base della programmazione orientata agli oggetti (**OOP** – *Object Oriented Programming*) c'è l'**oggetto**. Oggetti che hanno le stesse caratteristiche sono descritti da una stessa **classe**.

Un **oggetto** è caratterizzato da un suo stato, determinato dai valori di alcuni attributi, proprietà, (i **campi** di una classe), e da certi comportamenti, azioni, (i **metodi** di una classe). La programmazione ad oggetti descrive le interazioni tra oggetti.

Esempio

stato

Nome
Razza
Peso
Età

Di solito le variabili che descrivono lo stato sono locali e non sono visibili fuori dell'oggetto (**data hiding**)

comportamento

Dorme molto
Graffia i mobili
Caccia i topi
Attacca gli altri gatti

Il comportamento è controllato da funzioni che agiscono sulle variabili di stato e si interfacciano con l'esterno (**encapsulation**)



In C++ la **classe** costituisce la base per la programmazione orientata agli oggetti (**OOP** – *Object Oriented Programming*).

La classe è usata per definire la natura di un oggetto ed è l'unità base di **incapsulamento** del C++. Una **classe** ha un proprio **nome** e contiene due tipi di membri: **campi** e **metodi**.

dichiarazione di classe

```
class class_name {
    // private data (campi) and functions (metodi)
    access_specifier:
    // data and functions
    access_specifier:
    // data and functions
    // ...
    access_specifier:
    // data and functions
} object_list; ← opzionale
```

access specifier

- **public**: funzioni e dati accessibili da altre parti del programma.
- **private**: funzioni e dati accessibili solo all'interno della classe.
- **protected**: funzioni e dati accessibili dalla classe e dalle sue sotto-classi.

Differenza tra **classi** e **struct**: per default, tutti i membri di una **classe** sono **privati** e tutti quelli di una **struct** sono **pubblici**.

I membri (dati o funzioni) sono accessibili mediante **.** oppure **->**

Esempi

objectID.memberID

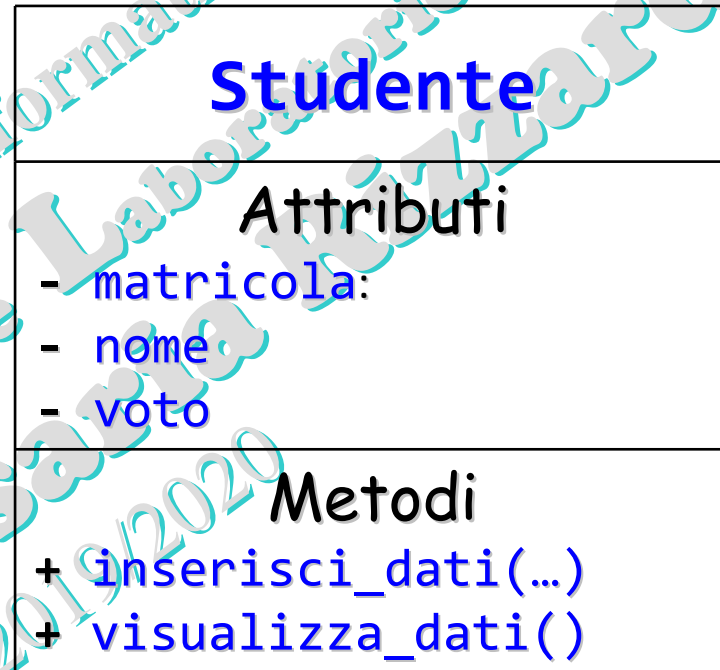
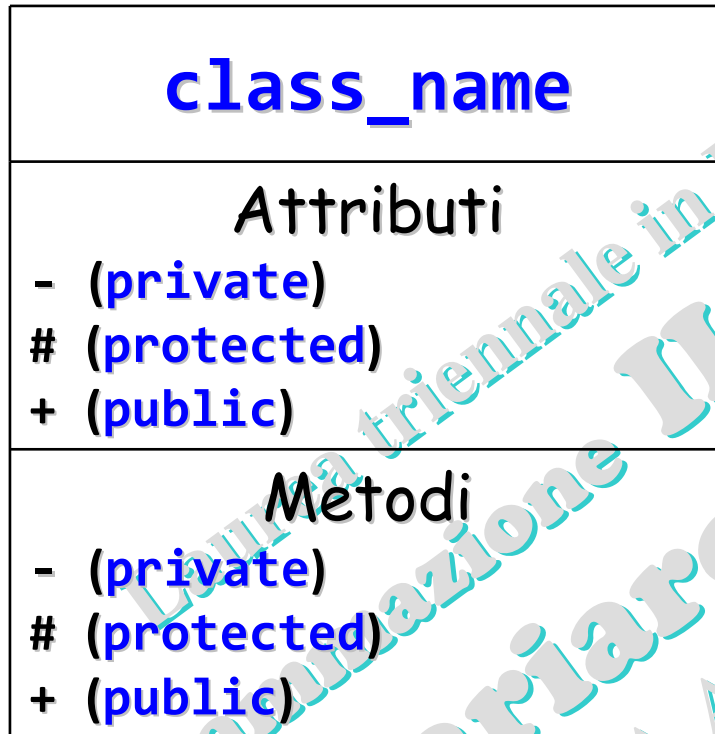
... oppure ...

objectID.classID::memberID

Access specifier

access specifier	public	protected	private
stessa classe	SI	SI	SI
classe derivata	SI	SI	NO
altre classi	SI	NO	NO

Diagramma di una classe



Esempio 1

Si vuole gestire un array di studenti memorizzati con i seguenti **campi**: **matricola** (10 char), **cognome e nome** (50 char), **voto esame** (unsigned int). **Operazioni**: inserimento dati dello studente, visualizzazione.

file **Studente.hpp** (specifica di classe)

```
class Studente {  
    char matr[11]; // 10+terminatore  
    char nome[51]; // 50+terminatore  
    unsigned int voto;  
  
public:  
    void inserisci_dati(const char *M, const char *N, unsigned int V);  
    void visualizza_dati( );  
};
```

attributi privati

metodi pubblici

file **Studente.cpp** (implementazione della classe)

```
#include <iostream>  
#include <cstring>  
#include "Studente.hpp"  
using namespace std;  
  
void Studente::inserisci_dati(const char *M, const char *N, unsigned int V)  
{  
    strcpy(matr,M);  
    strcpy(nome,N);  
    voto = V;  
}  
  
void Studente::visualizza_dati( )  
{  
    cout << "matr.: " << matr << "\tnome:" << nome  
        << "\tvoto: " << voto << endl;  
}
```

scope resolution operator

nome della classe

Esempio 1 (cont.)

file esempio1.cpp

```
#include "Studente.hpp"
```

```
#define NumStud 3
```

```
int main()  
{
```

```
    Studente E[NumStud];
```

```
    E[0].inserisci_dati("0124001233","Bianchi Aldo",26);
```

```
    E[1].inserisci_dati("0124001343","Rossi Maria",28);
```

```
    E[2].inserisci_dati("0124001345","Verdi Marco",24);
```

```
    for (int i=0; i<NumStud; i++)
```

```
        E[i].visualizza_dati();
```

```
    return 0;
```

```
}
```

istanzia un array di oggetti della classe

I metodi pubblici rappresentano
l'interfaccia con l'esterno

matr.: 0124001233

nome: Bianchi Aldo

voto: 26

matr.: 0124001343

nome: Rossi Maria

voto: 28

matr.: 0124001345

nome: Verdi Marco

voto: 24

Costruttori e distruttori

In una classe un **costruttore** è un metodo con lo stesso nome della classe che viene chiamato ogni volta che viene istanziato un oggetto della classe.

Un **costruttore** può essere usato per inizializzare i dati della classe. Per garantire che la memoria allocata dal costruttore sia deallocata c'è il **distruttore**, che ha sempre lo stesso nome della classe ma preceduto dal carattere "~".

Il costruttore e il distruttore non hanno valore di ritorno (neanche **void**).

Il distruttore non si può invocare da programma, è invocato dal compilatore quando viene deallocato lo spazio di memoria assegnato all'oggetto.

esempio2a.cpp

```
#include <iostream>
using namespace std;
class myclass {
public:
    int who;
    myclass(int id);
    ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id) {
    cout << "Initializing " << id << "\n";
    who = id;
}

myclass::~~myclass() {
    cout << "Destructing " << who << "\n";
} ...
```

Diagramma di annotazione:

- Un rettangolo con "costruttore" ha una freccia che punta a `myclass(int id);`.
- Un rettangolo con "distruttore" ha una freccia che punta a `~myclass();`.
- Un rettangolo con "globali" ha una freccia che punta a `glob_ob1(1), glob_ob2(2);`.

```
...
int main()
{
    myclass local_ob1(3);
    cout << "This will not be first line displayed.\n";
    myclass local_ob2(4);
    return 0;
}
```

Diagramma di annotazione:

- Un rettangolo con "locali" ha due frecce che puntano a `myclass local_ob1(3);` e `myclass local_ob2(4);`.

Output del programma:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1
```

Analisi dell'ordine di esecuzione:

- prima i globali e poi i locali**: Si riferisce alla fase di inizializzazione, dove prima vengono creati gli oggetti globali (1, 2) e poi i locali (3, 4).
- prima i locali e poi i globali**: Si riferisce alla fase di distruzione, dove prima vengono distrutti gli oggetti locali (4, 3) e poi i globali (2, 1).

Costruttore default

Se il programmatore non prevede alcun costruttore, il compilatore comunque crea il **costruttore default**.

Se il programmatore definisce un costruttore particolare, allora, **se richiesto**, va definito anche il costruttore default.

esempio2b.cpp

```
#include <iostream>
using namespace std;
class myclass {
public:
    int who;
    myclass(int id);    costruttore
    ~myclass();          con parametro
} glob_ob1(1), glob_ob2(2);
...
```

```
int main()
{
    myclass local_ob0;
    myclass local_ob1(3);
    ...
}
```

error: no matching function
for call to 'myclass::myclass()'

```
#include <iostream>
using namespace std;
class myclass {
public:
    int who;
    myclass() {};    costruttore
    myclass(int id); default
    ~myclass();
} glob_ob1(1), glob_ob2(2);
...
```

```
int main()
{
    myclass local_ob0;
    myclass local_ob1(3);
    cout << "This will not be first line displayed.\n";
    myclass local_ob2(4);
    return 0;
}
```

No error

Costruttore di copia

In una classe il **costruttore di copia** crea un oggetto a partire da un altro oggetto della classe (i valori delle variabili membro vengono copiati).

La sintassi dei costruttori di copia è la seguente:

```
class MyClass {  
    ...  
    MyClass(const MyClass &object);  
    ...  
};
```

Esempio

```
class Array {  
    public:  
        Array();  
        Array(int id);  
        Array(const Array &a);  
        ~Array();  
    ...  
    private:  
        int *arr;  
        int num;  
        int size;  
};
```

file di specifica di classe

```
#define MaxSIZE 100  
Array::Array()  
{  
    arr = new int[MaxSIZE];  
    num = 0;  
    size = maxSIZE;  
}  
  
Array::Array(int dim)  
{  
    arr = new int[dim];  
    num = 0;  
    size = dim;  
}  
...
```

```
Array::Array(const Array &s)  
{  
    num = s.num;  
    size = s.size;  
    arr = new int[size];  
    for (int i=0; i<num; i++)  
        arr[i] = s.arr[i];  
}  
  
Array::~~Array()  
{  
    delete[] arr;  
}
```

file implementazione di classe

function overloading

Function overloading indica la possibilità di usare lo stesso nome per due o più funzioni nella stessa classe. Ciò è possibile solo quando il compilatore è in grado di stabilire quale funzione chiamare e quindi le ridefinizioni di una funzione devono usare

- tipi differenti di parametri;
- numero diverso di parametri.

esempio3.cpp

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

int f(int i);
double f(double i); // tipi differenti di parametri
int f(int i, int j); // numero diverso di param.

int main()
{
    int h;
    double hh;
    h=f(10);
    h=f(10,20);
    hh=f(12.34);

    return 0;
}
```

```
1) int f(int i), i=10
3) int f(int i, int j), i=10, j=20
2) double f(double i), i=12.34
```

```
...
int f(int i)
{
    cout<<"1) int f(int i), i=" << i << endl;
    return i;
}


double f(double i)
{
    cout<<"2) double f(double i), i="<<i<<endl;
    return i;
}

int f(int i, int j)
{
    cout<<"3) int f(int i, int j), i="<<i<< ",
    j="<<j<<endl;
    return i+j;
}
```

Ereditarietà (inheritance)

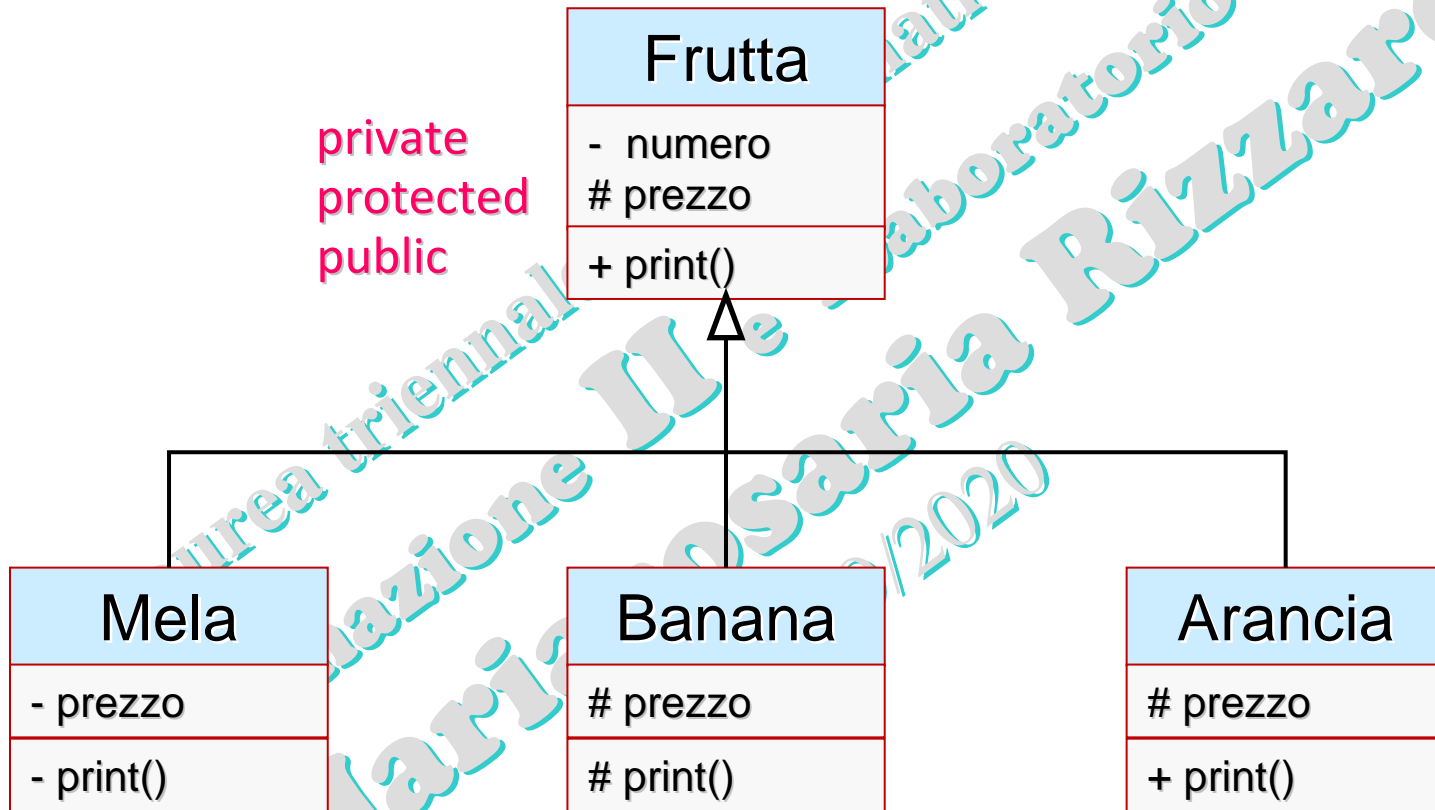
Una classe può essere derivata da un'altra classe: si instaura una **gerarchia** tra la **superclasse** (o **classe base**) e la **sottoclasse** (o **classe derivata**), che aggiunge dettagli. Sintassi:

```
class nomeClasseDerivata : access nomeClasseBase {  
    // corpo della classe derivata  
};
```



tipo di ereditarietà	membri della classe base	membri ereditati in classe derivata
public	public protected private	public protected inaccessibile
protected	public protected private	protected protected inaccessibile
private	public protected private	private private inaccessibile

Esempi di tipo di ereditarietà



Tipo di ereditarietà:
private

Tipo di ereditarietà:
protected

Tipo di ereditarietà:
public

Ereditarietà: esempio

```
class veicolo {  
    int numRuote;  
    int numPasseggeri;  
public:  
    void set_ruote(int n) {numRuote=n;}  
    int get_ruote() {return numRuote;}  
    void set_pass(int n) {numPasseggeri=n;}  
    int get_pass() {return numPasseggeri;}  
}
```

attributi privati comuni

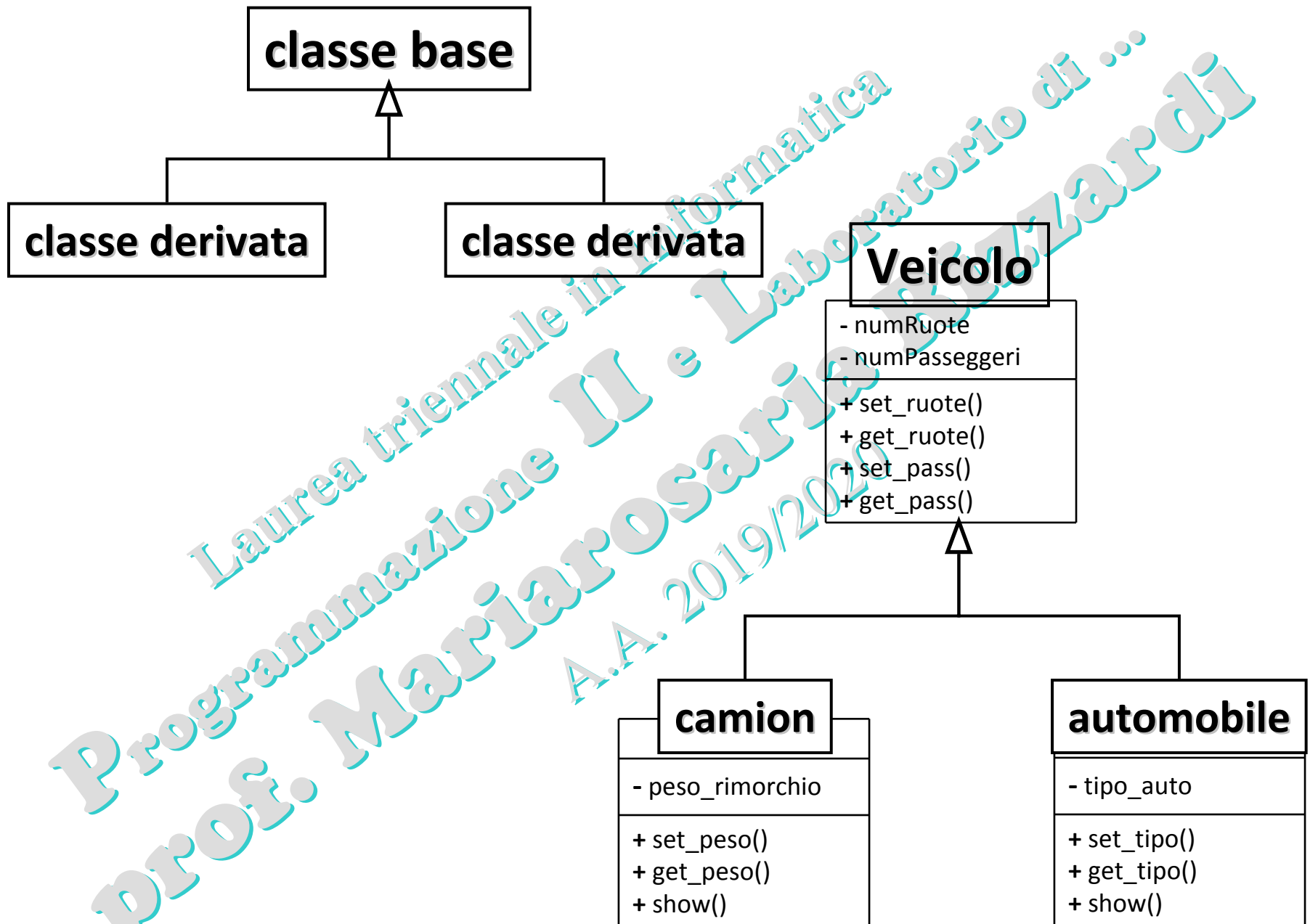
↑
classe base

classi derivate →

```
class camion : public veicolo {  
    int peso_rimorchio;  
public:  
    void set_peso(int w) {peso_rimorchio=w;}  
    int get_peso() {return peso_rimorchio;}  
}  
  
enum tipo {utilitaria, berlina, suv};  
class auto : public veicolo {  
    enum tipo tipo_auto;  
public:  
    void set_tipo(enum tipo t) {tipo_auto=t;}  
    enum tipo get_tipo() {return tipo_auto;}  
}
```

Laurea triennale
Programmatore II e
Prof. M. Rizzardi
A.A.

Diagramma delle classi



esempio4a.cpp (codice completo)

Veicolo.hpp

```
class veicolo {
    int numRuote;
    int numPasseggeri;
public:
    void set_ruote(int n) {numRuote=n;}
    int get_ruote() {return numRuote;}
    void set_pass(int n) {numPasseggeri=n;}
    int get_pass() {return numPasseggeri;}
};

class camion : public veicolo {
    int peso_rimorchio;
public:
    void set_peso(int w) {peso_rimorchio=w;}
    int get_peso() {return peso_rimorchio;}
    void show();
};

enum tipo {utilitaria, berlina, suv};
class automobile : public veicolo {
    enum tipo tipo_auto;
public:
    void set_tipo(enum tipo t) {tipo_auto=t;}
    enum tipo get_tipo() {return tipo_auto;}
    void show();
};
```

Veicolo_a.cpp

```
#include <iostream>
#include "Veicolo.hpp"
using namespace std;

void camion::show() {
    cout << "Camion" << endl;
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpeso: " << get_peso() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}

void automobile::show() {
    cout << "Auto" << endl;
    switch (get_tipo())
    {
        case utilitaria:
            cout << "\ttipo:  utilitaria\n";
            break;
        case berlina:
            cout << "\ttipo:  berlina\n";
            break;
        case suv:
            cout << "\ttipo:  suv\n";
            break;
    }
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}
```


esempio4a.cpp (cont.)

main.cpp

```
#include <iostream>
#include "Veicolo.hpp"

using namespace std;

int main()
{
    camion c1;
    c1.set_peso(3);
    c1.set_ruote(8);
    c1.set_pass(2);
    c1.show();

    cout << endl;
    automobile a1;
    a1.set_tipo(utilitaria);
    a1.set_pass(5);
    a1.set_ruote(4);
    a1.show();

    cout << endl;
    automobile a2;
    a2.set_tipo(suv);
    a2.set_pass(7);
    a2.set_ruote(4);
    a2.show();

    return 0;
}
```

Camion

ruote: 8
peso: 3
pass.: 2

Auto

tipo: utilitaria
ruote: 4
pass.: 5

Auto

tipo: suv
ruote: 4
pass.: 7

esempio4b.cpp (vector <string>)

Veicolo.hpp

```
class veicolo {
    int numRuote;
    int numPasseggeri;
public:
    void set_ruote(int n) {numRuote=n;}
    int get_ruote() {return numRuote;}
    void set_pass(int n) {numPasseggeri=n;}
    int get_pass() {return numPasseggeri;}
};

class camion : public veicolo {
    int peso_rimorchio;
public:
    void set_peso(int w) {peso_rimorchio=w;}
    int get_peso() {return peso_rimorchio;}
    void show();
};

enum tipo {utilitaria, berlina, suv};
class automobile : public veicolo {
    enum tipo tipo_auto;
public:
    void set_tipo(enum tipo t) {tipo_auto=t;}
    enum tipo get_tipo() {return tipo_auto;}
    void show();
};
```

Veicolo_b.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include "Veicolo.hpp"
using namespace std;

void camion::show() {
    cout << "Camion" << endl;
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpeso: " << get_peso() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}

void automobile::show() {
    vector<string> t;
    t.push_back("utilitaria");
    t.push_back("berlina");
    t.push_back("suv");
    cout << "Auto" << endl;
    cout << "\ttipo: " << t.at(get_tipo())
    << endl;
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}
```

t.at(j) equivalente a

```
cout << "\ttipo: " << t[(get_tipo())] << endl;
```

t[j]: come una componente di array

esempio4b.cpp (cont.)

main.cpp

```
#include <iostream>
#include "Veicolo.hpp"

using namespace std;

int main()
{
    camion c1;
    c1.set_peso(3);
    c1.set_ruote(8);
    c1.set_pass(2);
    c1.show();

    cout << endl;
    automobile a1;
    a1.set_tipo(utilitaria);
    a1.set_pass(5);
    a1.set_ruote(4);
    a1.show();

    cout << endl;
    automobile a2;
    a2.set_tipo(suv);
    a2.set_pass(7);
    a2.set_ruote(4);
    a2.show();

    return 0;
}
```

Camion

ruote: 8
peso: 3
pass.: 2

Auto

tipo: utilitaria
ruote: 4
pass.: 5

Auto

tipo: suv
ruote: 4
pass.: 7

esempio4c.cpp (con puntatore)

Veicolo_c.hpp

```
class veicolo {
    int numRuote;
    int numPasseggeri;
public:
    void set_ruote(int n) {numRuote=n;}
    int get_ruote() {return numRuote;}
    void set_pass(int n) {numPasseggeri=n;}
    int get_pass() {return numPasseggeri;}
    void show(); ← aggiunto a classe base
};

class camion : public veicolo {
    int peso_rimorchio;
public:
    void set_peso(int w) {peso_rimorchio=w;}
    int get_peso() {return peso_rimorchio;}
    void show();
};

enum tipo {utilitaria, berlina, suv};
class automobile : public veicolo {
    enum tipo tipo_auto;
public:
    void set_tipo(enum tipo t) {tipo_auto=t;}
    enum tipo get_tipo() {return tipo_auto;}
    void show();
};
```

Veicolo_c.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include "Veicolo_c.hpp"
using namespace std;

void veicolo::show() {
    cout << "Veicolo (classe base)" << endl;
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}

void camion::show() {
    cout << "Camion" << endl;
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpeso: " << get_peso() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}

void automobile::show() {
    vector<string> t;
    t.push_back("utilitaria");
    t.push_back("berlina");
    t.push_back("suv");
    cout << "Automobile" << endl;
    cout << "\ttipo: " << t.at(get_tipo())
    << endl;
    cout << "\truote: " << get_ruote() << endl;
    cout << "\tpass.: " << get_pass() << endl;
}
```

esempio4c.cpp (cont.)

main_c.cpp

```
#include <iostream>
#include "Veicolo_c.hpp"
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    veicolo v1, *p;
    p=&v1;
    p->set_ruote(2);
    p->set_pass(2);
    p->show();
```

p è un puntatore
ad un oggetto
della classe base

```
    cout << endl;
    camion c1;
    c1.set_peso(3);
    c1.set_ruote(8);
    c1.set_pass(2);
    c1.show();
```

```
    cout << endl;
    automobile a1;
    a1.set_tipo(utilitaria);
    a1.set_pass(5);
    a1.set_ruote(4);
    a1.show();
```

```
    return 0;
```

```
}
```

Veicolo (classe base)

ruote:	2
pass.:	2

Camion

ruote:	8
peso:	3
pass.:	2

Automobile

tipo:	utilitaria
ruote:	4
pass.:	5

esempio4d1.cpp (errore di compilazione)

main_d1.cpp

```
#include <iostream>
#include "Veicolo_c.hpp"
```

```
using namespace std;
```

```
int main()
{
```

```
    veicolo v1, *p;
    p=&v1;
    p->set_ruote(2);
    p->set_pass(2);
    p->show();
```

p è un puntatore ad
un oggetto della
classe base

```
    cout << endl;
```

```
    camion c1;
```

```
    p=&c1; p punta ad un oggetto della classe derivata
```

```
    p->set_peso(3);
```

error: 'class veicolo' has no member named 'set_peso'

```
    p->set_ruote(8);
```

```
    p->set_pass(2);
```

```
    p->show();
```

```
    cout << endl;
```

```
    automobile a1;
```

```
    a1.set_tipo(utilitaria);
```

```
    a1.set_pass(5);
```

```
    a1.set_ruote(4);
```

```
    a1.show();
```

```
    return 0;
```

```
}
```


esempio4d2.cpp (upcast)

main_d2.cpp

```
#include <iostream>
#include "Veicolo_c.hpp"
```

```
using namespace std;
```

```
int main()
{
```

```
    veicolo v1, *p;
    p=&v1;
    p->set_ruote(2);
    p->set_pass(2);
    p->show();
```

p è un puntatore
ad un oggetto
della classe base

```
    cout << endl;
    camion c1;
    p=&c1;
    //p->set_peso(3); //errore!
    p->set_ruote(8);
    p->set_pass(2);
    p->show();
```

"upcast" a veicolo

```
    cout << endl;
    automobile a1;
    a1.set_tipo(utilitaria);
    a1.set_pass(5);
    a1.set_ruote(4);
    a1.show();
```

```
    return 0;
```

```
}
```

In generale un puntatore ad un certo tipo non può puntare ad un oggetto di tipo diverso, con l'eccezione delle classi derivate.

Veicolo (classe base)
ruote: 2
pass.: 2

Veicolo (classe base)
ruote: 8
pass.: 2

Automobile
tipo: utilitaria
ruote: 4
pass.: 5

Nonostante `p=&c1` punti a un oggetto della classe `camion`, `p->show()` usa il metodo `show()` della classe base e NON della classe derivata `camion`

Upcasting

- Un puntatore ad una classe base può anche essere utilizzato come puntatore ad un oggetto di una qualunque classe derivata da tale classe base.
- Puntando ad un oggetto della classe derivata con un puntatore alla classe base si può accedere solo ai membri della classe base che sono stati ereditati dalla classe derivata.
- Non vale il viceversa: un puntatore ad una classe derivata non può puntare ad un oggetto della classe base.

esempio5a.cpp

```
class Base {
protected:
    int i;
public:
    void set_i(int a) {i=a;}
    int get_i() {return i;}
};

class Derived : public Base {
    int j;
public:
    void set_j(int a) {i=a;}
    int get_j() {return i;}
};
```

```
#include <iostream>
#include "es5.hpp"
using namespace std;

int main()
{
    Base *bp;
    Derived d;
    bp = &d;

    bp->set_i(10);
    cout << bp->get_i() << endl;

    bp->set_j(1);
    cout << bp->get_j() << endl;

    return 0;
}
```

OK!

ERRORE!

cast

CORREZIONE!

```
((Derived *)bp)->set_j(1);
cout << ((Derived *)bp)->get_j() << endl;
```

esempio5b.cpp

senza errori di compilazione, ma semanticamente sbagliato!

```
class Base {
protected:
    int i;
public:
    void set_i(int a) {i=a;}
    int get_i() {return i;}
};

class Derived : public Base {
    int j;
public:
    void set_j(int a) {i=a;}
    int get_j() {return i;}
};
```

```
#include <iostream>
#include "es5.hpp"
using namespace std;

int main()
{
    cout<<"sizeof(Base)    ="<<(int)sizeof(Base)<<endl;
    cout<<"sizeof(Derived)="<<(int)sizeof(Derived)<<endl;

    Derived d[2];
    d[0].set_i(1);
    d[1].set_i(2);

    Base *bp;
    bp = d;
    cout << "1: " << bp->get_i() << endl;

    bp++;
    cout << "2: " << bp->get_i() << endl;

    return 0;
}
```

```
sizeof(Base)    = 4
sizeof(Derived) = 8
1: 1
2: 0
```

bp=bp+sizeof(...)

????

Virtual functions e polymorphism

Una **funzione virtuale** è una funzione membro dichiarata in una classe base e ridefinita in una classe derivata. Le funzioni virtuali sono lo strumento principale del **polimorfismo** ed implementano la filosofia “*una sola interfaccia, più metodi*”.

esempio6.cpp

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

class base {
public:
    virtual void vfunc() {
        cout << "metodo vfunc() della classe base\n";
    }
};

class derived1 : public base {
public:
    void vfunc() {
        cout << "metodo vfunc() della classe derived1\n";
    }
};

class derived2 : public base {
public:
    void vfunc() {
        cout << "metodo vfunc() della classe derived2\n";
    }
};
```

```
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;

    p = &b;
    p->vfunc();

    p=&d1;
    p->vfunc();

    p=&d2;
    p->vfunc();
}
```

p è un puntatore ad un oggetto della classe base

p punta ad un oggetto della classe derivata

metodo vfunc() della classe base
metodo vfunc() della classe derived1
metodo vfunc() della classe derived2

Esercizio: riscrivere l'**esempio 4d2** con funzione virtuale **show()**

overload vs override

Function overloading (*compile time polymorphism*) indica la possibilità di usare lo **stesso nome** per due o più funzioni nella stessa classe. Ciò è possibile solo quando la “*signature*” (tipo e numero dei parametri) delle funzioni è diversa.

esempio3.cpp

```
int f(int i);
double f(double i);
int f(int i, int j);

int main()
{   int h; double hh;
    h=f(10);
    h=f(10,20);
    hh=f(12.34);
    return 0;
}
```

Function overriding (*run-time polymorphism*) indica la possibilità per una **classe derivata** di ridefinire una funzione della superclasse con la stessa “*signature*”, cioè eguale tipo dei parametri e del valore di ritorno.

esempio6.cpp

```
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;

    p = &b;
    p->vfunc();

    p=&d1;
    p->vfunc();

    p=&d2;
    p->vfunc();
}
```

(puntatore a funzione virtuale in classe base)

I costruttori possono essere
“overloaded”, ma non possono essere

function overload vs function override

- ❖ **Ereditarietà**: l'**overriding** di funzioni si verifica solo quando una classe eredita da un'altra classe, mentre l'**overloading** può avvenire anche senza ereditarietà.
- ❖ **Function Signature**: Le funzioni "sovraccaricate" ("overloaded") devono differire in "signature", cioè o nel numero dei parametri oppure nel tipo dei parametri; invece nell'"**overriding**" la "signature" delle funzioni deve essere la stessa.
- ❖ **"Scope" delle funzioni**: le funzioni "**overridden**" hanno "scope" (ambiti di visibilità) differenti; mentre le funzioni "**overloaded**" hanno lo stesso "scope".
- ❖ **Comportamento delle funzioni**: l'**overriding** è necessario quando una funzione della classe derivata deve avere qualche funzionalità aggiuntiva oppure diversa da quella della classe base. L'**overloading** è usato per avere funzioni con lo stesso nome che si comportano in modo diverso in dipendenza dei parametri passati ad esse.

Pure virtual function e abstract class

Una **funzione virtuale pura** è una funzione virtuale che non ha una definizione nella classe base, la quale viene detta **classe base astratta**.

Una classe è **astratta** se contiene almeno una funzione virtuale pura.

La sintassi per dichiarare una **funzione virtuale pura** è:

```
virtual type func-name(parameter-list) = 0;
```

In tal caso ciascuna classe derivata deve prevedere la sua propria definizione altrimenti verrà considerata anch'essa **classe astratta**. Se la classe derivata non riesce a sovrascrivere la funzione virtuale pura, si verificherà un errore in fase di compilazione. Non si può istanziare un oggetto di una classe astratta.

esempio7a.cpp

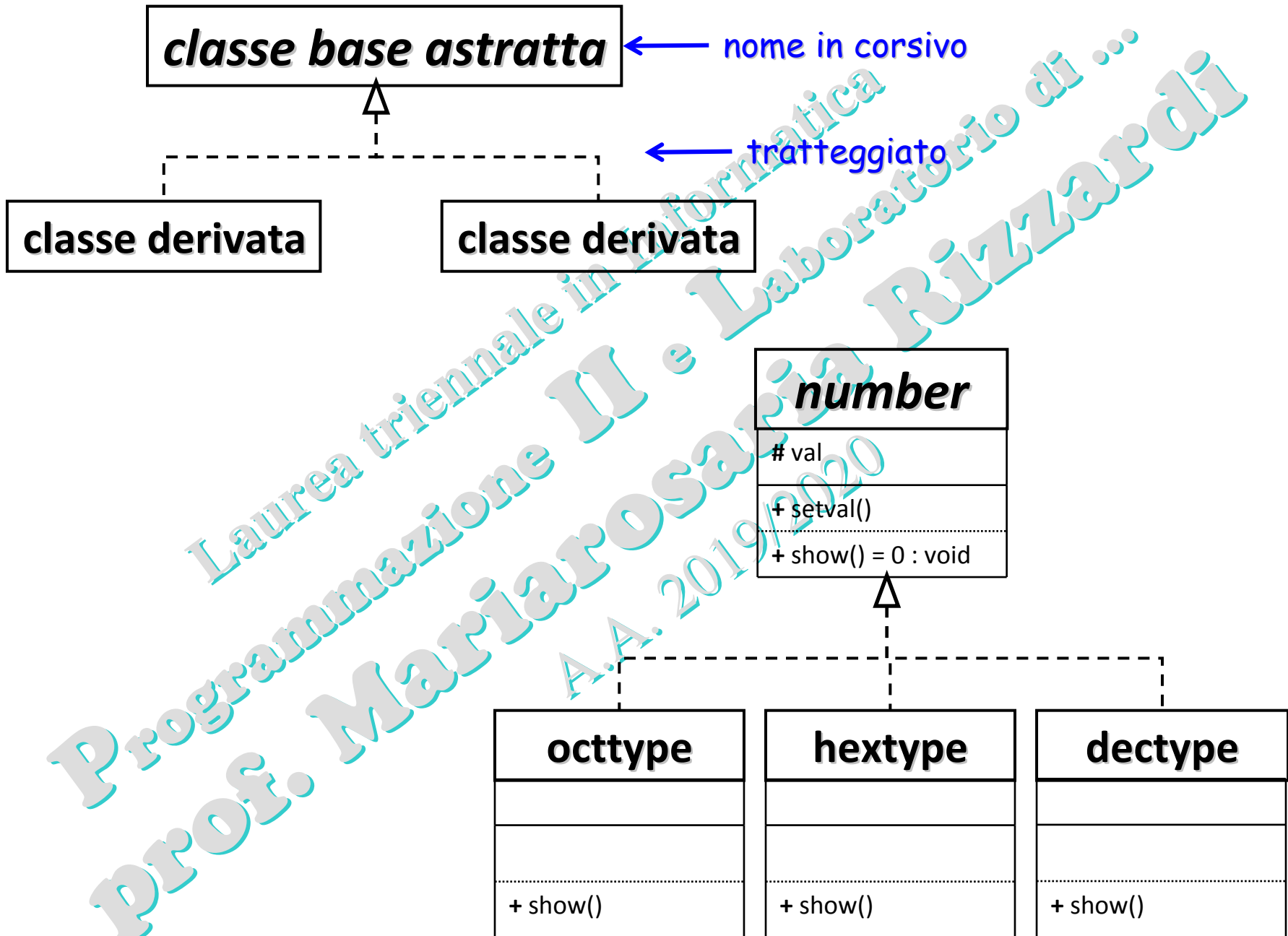
```
class number {  
    protected: classe astratta  
    int val;  
    public:  
        void setval(int i) { val = i; }  
        virtual void show() = 0;  
}; funzione virtuale pura  
  
class hextype : public number {  
    public:  
        void show() {  
            cout << "hex.: ";  
            cout << hex << val << "\n";  
        }  
};
```

```
class decatype : public number {  
    public:  
        void show() {  
            cout << "dec.: ";  
            cout << val << "\n";  
        }  
};  
  
class octtype : public number {  
    public:  
        void show() {  
            cout << "oct.: ";  
            cout << oct << val << "\n";  
        }  
};
```

```
int main()  
{  
    decatype d;  
    hextype h;  
    octtype o;  
    d.setval(20);  
    d.show();  
    h.setval(20);  
    h.show();  
    o.setval(20);  
    o.show();  
    return 0;  
}
```

```
dec.: 20  
hex.: 14  
oct.: 24
```

Diagramma delle classi



```
class number {
protected:
    int val;
public:
    void setval(int i) { val = i; }
    virtual void show() = 0;
};
```

classe astratta

```
class hextype : public number {
public:
    void show() {
        cout << "hex.: ";
        cout << hex << val << "\n";
    }
};
```

```
int main()
{
    number n;
    dectype d;
    hextype h;
    octtype o;

    n.setval(20);
    n.show();

    d.setval(20);
    d.show();

    h.setval(20);
    h.show();

    o.setval(20);
    o.show();

    return 0;
}
```

```
class dectype : public number {
public:
    void show() {
        cout << "dec.: ";
        cout << val << "\n";
    }
};

class octtype : public number {
public:
    void show() {
        cout << "oct.: ";
        cout << oct << val << "\n";
    }
};
```

```
>g++ -o main.exe esempio6.cpp
esempio6.cpp: In function 'int main()':
esempio6.cpp:43:13: error: cannot declare variable 'n' to be of abstract type 'number'
    number n;
    ^
esempio6.cpp:8:7: note: because the following virtual functions are pure within 'number':
    class number {
    ^~~~~~
esempio6.cpp:14:18: note: 'virtual void number::show()'
    virtual void show() = 0;
```

Non si può istanziare un oggetto di una classe astratta

Ereditarietà (inheritance)

Costruttori, distruttori, metodi virtuali non sono ereditati in una classe derivata.

Se necessari, questi vanno creati.

Laurea triennale in Informatica
Programmazione II e Laboratorio C++
prof. Mariarosaria Rizzardi
A.A. 2019/2020