

OOP

VARIE

:: E .

- ▶ In molti esempi, per accedere ad un elemento di un namespace è stato usato l'operatore " :: "
- ▶ Per accedere ad un elemento di una classe, invece, è stato usato l'operatore " . "
- ▶ Da StackOverflow:
 - ▶ *"Stavo usando la libreria xxx che aveva una classe che i membri invece di accederli con . li dovevi accedere con :: oh, minkia, lo giuro!"*
- ▶ :: è operatore di risoluzione di scope, e viene applicato anche a qualcosa che è definito all'interno dello scope di una classe

:: E .

► In breve:

► ::

- si usa lo **scope resolver operator** per accedere a qualcosa che appartiene a tutte le istanze di una classe

► .

- si usa il **member operator** per accedere a qualcosa che appartiene ad un oggetto

:: E .

- ▶ In breve:
- ▶ ::
 - si usa lo scope resolver operator per accedere a qualcosa che appartiene a **tutte le istanze di una classe**
- ▶ .
 - si usa il member operator per accedere a qualcosa che appartiene ad un oggetto
- ▶ Cosa appartiene a "tutte le istanze di una classe"?

:: E .

- ▶ In breve:
- ▶ ::
 - si usa lo scope resolver operator per accedere a qualcosa che appartiene a **tutte le istanze di una classe**
- ▶ .
 - si usa il member operator per accedere a qualcosa che appartiene ad un oggetto
- ▶ Cosa appartiene a "tutte le istanze di una classe"?
 - implementazioni di funzione
 - membri definiti come `static`
 - classi innestate (definite all'interno di altre classi)

:: E .

- ▶ `static` è la parola con più overload in assoluto in C++!
Oggi vediamo solo la definizione di membri static all'interno di una classe
- ▶ Membri definiti come static sono condivisi tra tutte le istanze di una classe

```
class GlobalCounter {  
public:  
    int increment() {  
        return ++count;  
    }  
private:  
    static int count;  
};  
  
int GlobalCounter::count = 0;  
  
int main() {  
    GlobalCounter gc1;  
    GlobalCounter gc2;  
    // Prints 1  
    cout << gc1.increment() << endl;  
    // Prints 2  
    cout << gc2.increment() << endl;  
}
```

:: E .

- ▶ `static` è la parola con più overload in assoluto in C++!
Oggi vediamo solo la definizione di membri static all'interno di una classe
- ▶ Membri definiti come static sono condivisi tra tutte le istanze di una classe

```
class GlobalCounter {  
public:  
    int increment() {  
        return ++count;  
    }  
private:  
    static int count;  
};  
  
int GlobalCounter::count = 0;  
  
int main() {  
    GlobalCounter gc1;  
    GlobalCounter gc2;  
    // Prints 1  
    cout << gc1.increment() << endl;  
    // Prints 2  
    cout << gc2.increment() << endl;  
}
```

- ▶ NOTA: l'esempio è corretto anche se `count` è dichiarata con il qualificatore `private`.
Lo standard prevede che le variabili `static` debbano essere dichiarate ed inizializzate a livello globale. Ogni altro accesso alla var `count` genera errore di compilazione.

:: E .

- Occhio: questa non compila!

```
class GlobalCounter {  
public:  
    int increment() {  
        return ++count;  
    }  
private:  
    static int count = 0;  
};
```

- Ma questa sì:

```
class Angle {  
public:  
    // TODO: Implement actual functions  
private:  
    const static double PI = 3.14159;  
    double angle;  
};
```

:: E .

► Per quanto detto prima, per accedere ai membri statici uso l'operatore ::

```
► class Day {  
    public:  
        const static int NUMERO_DI_ORE = 24;  
        // altra roba  
    private:  
        // membri privati  
}  
  
int main() {  
    std::cout << Day::NUMERO_DI_ORE << std::endl;  
}
```

:: E .

- ▶ Per quanto detto prima, per accedere ai membri statici uso l'operatore ::

```
▶ class Day {  
    public:  
        const static int NUMERO_DI_ORE = 24;  
        // altra roba  
    private:  
        // membri privati  
}  
  
int main() {  
    std::cout << Day::NUMERO_DI_ORE << std::endl;  
}
```

- ▶ Non manca qualcosa?

:: E .

► Esistono anche funzioni membro di tipo static!

```
► class Day {  
    public:  
        static int numeroOre() { return 24;}  
        // altra roba  
    private:  
        // membri privati  
}  
  
int main() {  
    std::cout << Day::numeroOre() << std::endl;  
}
```

:: E .

- ▶ Esistono anche funzioni membro di tipo static!

```
▶ class Day {  
    public:  
        static int numeroOre() { return 24;}  
        // altra roba  
    private:  
        // membri privati  
}  
  
int main() {  
    std::cout << Day::numeroOre() << std::endl;  
}
```

- ▶ Anche qui manca qualcosa!!!!

:: E .

- ▶ Dietro le quinte: i membri di tipo `static` sono istanziati solo una volta all'inizio dell'esecuzione del programma e vengono distrutti solo al termine di esso.
- ▶ Per questo motivo, i membri `static` NON sono associati con gli oggetti di una classe: è meglio pensarli come appartenenti alla classe stessa, piuttosto che alle sue istanze.
- ▶ Non necessitano di costruttore!
- ▶ L'utilizzo di funzioni membro qualificate con `static` è di norma considerata una cattiva pratica. Ancora peggio è la scrittura di classi aventi solamente membri e funzioni membro `static`: equivale a dichiarare funzioni e variabili globali, con tutti gli svantaggi che ne derivano!

OVERLOAD DEGLI OPERATORI

- ▶ Solo accenno:
tutti gli operatori possono essere visti come delle funzioni
- ▶ Pensate a operatore " + " che opera su valori interi come una funzione che prende in ingresso due valori e ne restituisce un terzo pari alla somma dei due
- ▶ Oppure, operatore " [] " che opera su puntatori equivale a dereferenziatore *
$$p[3] = *(p + 3)$$
- ▶ Costruiamo una funzione che realizzi l'operatore " ++ postfix " su interi

```
int operator++( int& i ) {  
  
}
```

OVERLOAD DEGLI OPERATORI

- ▶ Solo accenno:
tutti gli operatori possono essere visti come delle funzioni
- ▶ Pensate a operatore " + " che opera su valori interi come una funzione che prende in ingresso due valori e ne restituisce un terzo pari alla somma dei due
- ▶ Oppure, operatore " [] " che opera su puntatori equivale a dereferenziatore *
$$p[3] = *(p + 3)$$
- ▶ Costruiamo una funzione che restituisca un valore incrementato su interi

```
int operator++( int& i ) {  
    return i = i + 1;  
}
```

- ▶ // Non l'ho testato... È corretto?

OVERLOAD DEGLI OPERATORI

- ▶ Abbiamo visto che però in l'operatore " + " applicato su due oggetti di tipo `std::string` esegue l'operazione di concatenazione
- ▶ Quando definiamo una classe, abbiamo la possibilità di ridefinire le funzionalità di un mucchio di operatori di linguaggio, in modo da adattarli al tipo di dato che la classe modella.

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- ▶ Nota: questi, invece,

::	.*	.	?:
----	----	---	----

 no

OVERLOAD DEGLI OPERATORI

- Tra gli operatori che possiamo ridefinire, ne spicca uno in particolare...

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

OVERLOAD DEGLI OPERATORI

- ▶ Tra gli operatori che possiamo ridefinire, ne spicca uno in particolare...

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- ▶ Operatore " () ": chiamata a funzione
- ▶ L'overload di questo operatore definisce un oggetto che lavora come una funzione; questi oggetti sono chiamati **functor** o **function object**

OVERLOAD DEGLI OPERATORI

- Vantaggio degli oggetti funzione rispetto alle funzioni standard: mantengono lo stato o i valori tra una chiamata e l'altra

```
1  class Accumulator
2  {
3  private:
4      int m_counter = 0;
5
6  public:
7      Accumulator()
8      {
9      }
10
11     int operator() (int i) { return (m_counter += i); }
12 };
13
14 int main()
15 {
16     Accumulator acc;
17     std::cout << acc(10) << std::endl; // prints 10
18     std::cout << acc(20) << std::endl; // prints 30
19
20     return 0;
21 }
```

OVERLOAD DEGLI OPERATORI

- Tutto qui? No, overload di () rende l'oggetto callable

```
// constructing sets
#include <iostream>
#include <set>

bool fncomp (int lhs, int rhs) {return lhs<rhs;}

struct classcomp {
    bool operator() (const int& lhs, const int& rhs) const
    {return lhs<rhs;}
};

int main ()
{
    std::set<int> first;                // empty set of ints

    int myints[] = {10,20,30,40,50};
    std::set<int> second (myints,myints+5);    // range

    std::set<int> third (second);           // a copy of second

    std::set<int> fourth (second.begin(), second.end()); // iterator ctor.

    std::set<int,classcomp> fifth;          // class as Compare

    bool (*fn_pt) (int,int) = fncomp;
    std::set<int,bool(*) (int,int)> sixth (fn_pt); // function pointer as Compare

    return 0;
}
```

OOP

OOP

- ▶ Scrivere un programma secondo paradigma di programmazione procedurale / imperativo (C) prevede:
 - dividere il programma in una serie di task e sotto-task
 - scrivere delle funzioni per ogni task individuato
 - far eseguire in sequenza da un computer i task opportunamente preparati
- ▶ Complessità del programma cresce al crescere del numero dei task e della complessità del problema da risolvere.
- ▶ Più opportuno introdurre astrazione in questo modello...

Consider the task of modeling the operation of a car. Such a program would have lots of separate variables storing information on various car parts, and there'd be no way to group together all the code that relates to, say, the wheels. It's hard to keep all these variables and the connections between all the functions in mind.

To manage this complexity, it's nicer to package up self-sufficient, modular pieces of code. People think of the world in terms of interacting *objects*: we'd talk about interactions between the steering wheel, the pedals, the wheels, etc. OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

OOP

- ▶ Esistono una miriade di definizioni di programmazione orientata ad oggetti, ma tutte citano i tre concetti base:
- ▶ - Incapsulamento:

OOP

- ▶ Esistono una miriade di definizioni di programmazione orientata ad oggetti, ma tutte citano i tre concetti base:
- ▶ - Incapsulamento: i dati che definisco un oggetto e i metodi che definiscono la logica sono accessibili dai membri dell'oggetto stesso, ma non sono visibili all'esterno. Per poter interagire si usano metodi pubblici (oggetto come black-box)
- ▶ - Ereditarietà:

OOP

- ▶ Esistono una miriade di definizioni di programmazione orientata ad oggetti, ma tutte citano i tre concetti base:
- ▶ - Incapsulamento: i dati che definiscono un oggetto e i metodi che definiscono la logica sono accessibili dai membri dell'oggetto stesso, ma non sono visibili all'esterno. Per poter interagire si usano metodi pubblici (oggetto come black-box)
- ▶ - Ereditarietà: principio per cui è possibile derivare nuove classi a partire da quelle già definite, realizzando una gerarchia. Sottoclassi derivate mantengono metodi e membri della super classe da cui derivano, ma possono anche ridefinirli (over-riding) o aggiungere nuovi metodi e membri
- ▶ - Polimorfismo:

OOP

- ▶ Esistono una miriade di definizioni di programmazione orientata ad oggetti, ma tutte citano i tre concetti base:
- ▶ - Incapsulamento: i dati che definisco un oggetto e i metodi che definiscono la logica sono accessibili dai membri dell'oggetto stesso, ma non sono visibili all'esterno. Per poter interagire si usano metodi pubblici (oggetto come black-box)
- ▶ - Ereditarietà: principio per cui è possibile derivare nuove classi a partire da quelle già definite, realizzando una gerarchia. Sottoclassi derivate mantengono metodi e membri della super classe da cui derivano, ma possono anche ridefinirli (over-riding) o aggiungere nuovi metodi e membri
- ▶ - Polimorfismo: viene permesso ad un oggetto di essere uno tra un numero finito di tipi, ognuno che implementa versioni diverse di uno stesso metodo; la decisione su quale metodo eseguire verrà presa a runtime, a seconda del tipo di dato che viene costruito

OOP

- ▶ Esistono una miriade di definizioni di programmazione orientata ad oggetti, ma tutte citano i tre concetti base:
- ▶ - Incapsulamento: i dati che definisco un oggetto e i metodi che definiscono la logica sono accessibili dai membri dell'oggetto stesso, ma non sono visibili all'esterno. Per poter interagire si usano metodi pubblici (oggetto come black-box)
- ▶ - Ereditarietà: principio per cui è possibile derivare nuove classi a partire da quelle già definite, realizzando una gerarchia. Sottoclassi derivate mantengono metodi e membri della super classe da cui derivano, ma possono anche ridefinirli (over-riding) o aggiungere nuovi metodi e membri
- ▶ - Polimorfismo: viene permesso ad un oggetto di essere uno tra un numero finito di tipi, ognuno che implementa versioni diverse di uno stesso metodo; la decisione su quale metodo eseguire verrà presa a runtime, a seconda del tipo di dato che viene costruito
- ▶ **Ne sapete sicuramente più di me.**

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Incapsulamento: va beh, con le classi...
- ▶ Contrariamente a quanto avviene in Java, non esiste la parola riservata `interface`. Il concetto di interfaccia viene gestito con membri pubblici e privati tramite le keyword `public`, `private` e `protected`

In Java:

```
interface Fooer {  
    public void foo();  
}  
  
class Bar implements Fooer {  
    public void foo() {  
        // implementation  
    }  
}
```

In C++:

```
class Fooer {  
public:  
    virtual void foo() = 0;  
};  
  
class Bar : public Fooer {  
public:  
    void foo() {  
        // implementation  
    }  
};
```

OOP

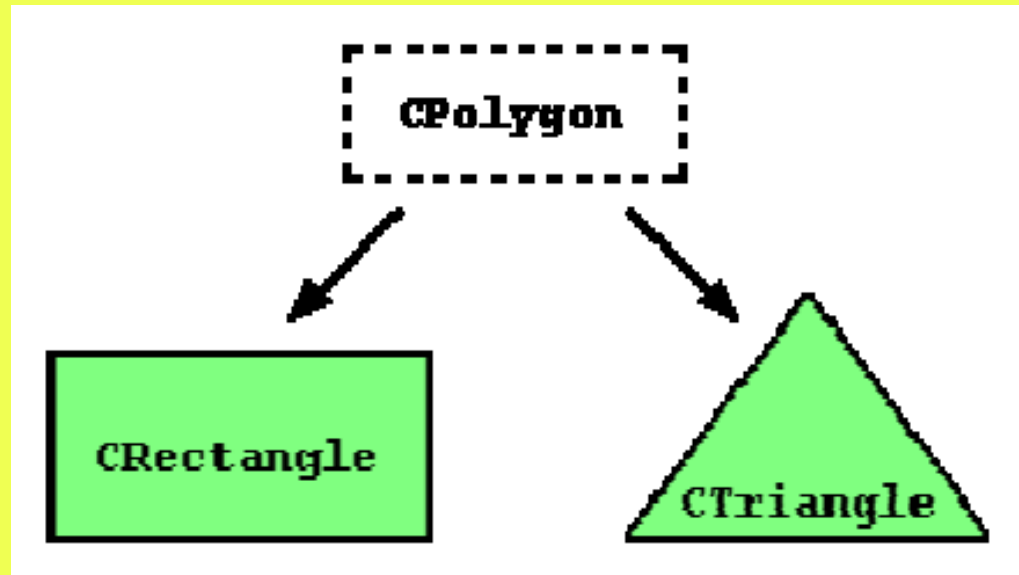
- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Incapsulamento: linee guida
- ▶ Come già ricordato, i dati *dovrebbero* essere resi privati. Per accedervi in lettura e scrittura *si dovrebbe* usare coppia di funzioni membro get/set.
- ▶ Le funzioni *dovrebbero* essere sia pubbliche che private:
 - funzioni pubbliche realizzano l'interfaccia
 - funzioni private usate solo internamente alla classe stessa

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà: permette di definire dei legami gerarchici tra classi che condividono stesse caratteristiche funzionali
- ▶ Facciamo l'esempio che fanno tutti...

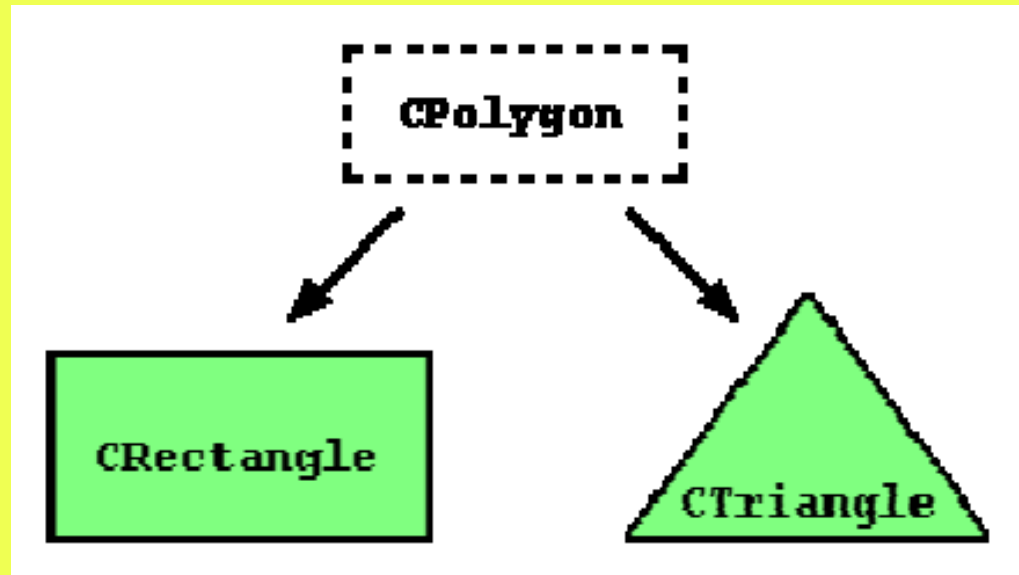
OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà: permette di definire dei legami gerarchici tra classi che condividono stesse caratteristiche funzionali
- ▶ Facciamo l'esempio che fanno tutti...



OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà: permette di definire dei legami gerarchici tra classi che condividono stesse caratteristiche funzionali
- ▶ Facciamo l'esempio che fanno tutti...



- ▶ Mi dispiace.

OOP

► Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?

► Ereditarietà:

```
1 class Vehicle {  
2     protected:  
3         string license;  
4         int year;  
5  
6     public:  
7         Vehicle(const string &myLicense, const int myYear)  
8             : license(myLicense), year(myYear) {}  
9         const string getDesc() const  
10            {return license + " from " + stringify(year);}  
11         const string &getLicense() const {return license;}  
12         const int getYear() const {return year;}  
13 };
```


OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà:

```
1 class Vehicle {
2     protected:
3         string license;
4         int year;
5
6     public:
7         Vehicle(const string &myLicense, const int myYear)
8             : license(myLicense), year(myYear) {}
9         const string getDesc() const
10             {return license + " from " + stringify(year);}
11         const string &getLicense() const {return license;}
12         const int getYear() const {return year;}
13 };
```

```
1 class Car : public Vehicle { // Makes Car inherit from Vehicle
2     string style;
3
4     public:
5         Car(const string &myLicense, const int myYear, const string
6             &myStyle)
7             : Vehicle(myLicense, myYear), style(myStyle) {}
8         const string &getStyle() {return style;}
9     };
10 };
```

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà:
- ▶ Affrontiamo gli specificatori di accesso `public`, `private` e `protected`.
- ▶ Membri **private** possono essere visti solo da entità definite all'interno della stessa classe
- ▶ Membri **protected** possono essere visti solo da entità definite all'interno della stessa classe e dalle sottoclassi derivate
- ▶ Membri **public** possono essere visti da qualunque altra entità
- ▶ Differenza non riguarda solo l'accesso ai membri, ma anche a cosa viene ereditato dalle sottoclassi (e come)
("E' una" vs. "Ha una"...) 

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà:

Private is the way to go if you just want the functionality
Public is the way to go otherwise

public inheritance: Is-A relationship (Student is a Person)

protected inheritance: Implemented-In-Terms-Of relationship, or Has-A relationship
(Car has an Engine)

private inheritance: Implemented-In-Terms-Of relationship, or Has-A relationship
(Car has an Engine)

- ▶ Consiglio: specificare sempre public. Se serve ereditarietà privata o protetta, forse la classe è pensata male. Composizione è la risposta!

No

```
class A : B {  
};
```

```
class A {  
private:  
    B b;  
};
```

Ok!

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Ereditarietà:

```
class CPolygon {  
protected:  
    int width, height;  
public:  
    void set_values (int a, int b)  
        { width=a; height=b;}  
};  
  
class CRectangle: public CPolygon {  
public:  
    int area ()  
        { return (width * height); }  
};  
  
class CTriangle: public CPolygon {  
public:  
    int area ()  
        { return (width * height / 2); }  
};  
  
int main () {  
    CRectangle rect;  
    CTriangle trgl;  
    rect.set_values (4,5);  
    trgl.set_values (4,5);  
    cout << rect.area() << endl;  
    cout << trgl.area() << endl;  
    return 0;  
}
```

OOP

- ▶ Veniamo ai fatti: la
- ▶ Ereditarietà:

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

ine della oop?

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:
- ▶ Overload dei membri funzione: metodi virtual nella classe base

```
class Employee {  
public:  
    Employee(const string& name, int dept);  
    virtual void print() const;  
    // ...  
private:  
    string first_name, family_name;  
    short department;  
    // ...  
};
```

```
class Manager : public Employee {  
public:  
    Manager(const string& name, int dept, int lvl);  
    void print() const;  
    // ...  
private:  
    list<Employee*> group;  
    short level;  
    // ...  
};  
  
void Manager::print() const  
{  
    Employee::print();  
    cout << "\tlevel " << level << '\n';  
    // ...  
}
```


OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:
- ▶ Overload dei membri funzione: metodi virtual nella classe base

```
class Employee {  
public:  
    Employee(const string& name, int dept);  
    virtual void print() const;  
    // ...  
private:  
    string first_name, family_name;  
    short department;  
    // ...  
};
```

```
void Employee::print() const  
{  
    cout << family_name << '\t' << department << '\n';  
    // ...  
}
```

- ▶ Nota: una funzione virtual deve essere definita nella classe base!
Altra nota: affinché la magia funzioni, segnatura di funzioni deve essere la stessa...
Altra nota ancora: funzioni virtuali pure realizzano metodi astratti (interfacce)

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:
- ▶ Vero polimorfismo si ottiene con puntatori:

```
class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
        { cout << this->area() << endl; }
};

class CRectangle: public CPolygon {
public:
    int area (void)
        { return (width * height); }
};
```

```
class CTriangle: public CPolygon {
public:
    int area (void)
        { return (width * height / 2); }
};

int main () {
    CPolygon * ppoly1 = new CRectangle;
    CPolygon * ppoly2 = new CTriangle;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    delete ppoly1;
    delete ppoly2;
    return 0;
}
```

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:

```
class Base {  
    virtual void foo();  
};  
  
class Derived : public Base {  
    void foo();  
};
```

```
class Base {  
    virtual void foo();  
};  
  
class Derived : public Base {  
    // Checks that this will override a function  
    void foo() override;  
};
```

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:

```
class Base {  
    virtual void foo();  
};  
  
class Derived : public Base {  
    void foo();  
};
```

```
class Base {  
};  
  
class Derived : public Base {  
    // Error  
    void foo() override;  
};
```

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:

```
class Base {  
    virtual void foo();  
};  
  
class Derived : public Base {  
    void foo();  
};
```

```
class Base {  
    virtual void foo() const;  
};  
  
class Derived : public Base {  
    // Error  
    void foo() override;  
};
```

OOP

- ▶ Veniamo ai fatti: lo standard C++ come affronta i tre principi cardine della oop?
- ▶ Polimorfismo:

What goes wrong if you forget `virtual`?

```
class Base {  
    void foo();  
    virtual void bar();  
};  
class Derived {  
    void foo();  
    void bar();  
};  
  
Derived *d = new Derived();  
Base *b = dynamic_cast<Base *>(d);  
d->foo(); // Calls Derived::foo  
d->bar(); // Calls Derived::bar  
b->foo(); // Calls Base::foo  
b->bar(); // Calls Derived::bar
```

In generale meglio non fare override di funzioni non virtual (non è illegale, solo cattiva abitudine...)

OOP

- ▶ Casino? Beh, guardiamo qualche esempio...



OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costruttore A " << a << std::endl; }  
    ~A()  
        { std::cout << "distruttore A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A a;  
}
```


OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costruttore A " << a << std::endl; }  
    ~A()  
        { std::cout << "distruttore A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A a;  
}
```

- costruttore A 17
distruttore A

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costruttore A " << a << std::endl; }  
    ~A()  
        { std::cout << "distruttore A" << std::endl; }  
    void stampa()  
        { std::cout << "stampa A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A a;  
    a.stampa();  
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costruttore A " << a << std::endl; }  
    ~A()  
        { std::cout << "distruttore A" << std::endl; }  
    void stampa()  
        { std::cout << "stampa A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A a;  
    a.stampa();  
}
```

- costruttore A 17
stampa A
distruttore A

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A *a = new A;  
    a->stampa();  
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A *a = new A;  
    a->stampa();  
}
```

- **A non viene distrutto automaticamente!**
- costruttore A 17
stampa A

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
int main() {  
    A *a = new A;  
    a->stampa();  
    delete a;  
}
```

- costr A 17
stmp A
distr A

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {
public:
    A() : a(17)
        { std::cout << "costr A " << a << std::endl; }
    ~A()
        { std::cout << "distr A" << std::endl; }
    void stampa()
        { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
        { std::cout << "costr B " << a << std::endl;}
    ~B()
        { std::cout << "distr B" << std::endl;}
    void stampa()
        { std::cout << "stmp B" << std::endl;}
};

int main() {
    A a;
    B b;
    a.stampa();
    b.stampa();
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- costr A 17
costr A 17
costr B 17
stmp A
stmp B
distr B
distr A
distr A

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
class B : public A {  
public:  
    B()  
        { std::cout << "costr B " << a << std::endl;}  
    ~B()  
        { std::cout << "distr B" << std::endl;}  
    void stampa()  
        { std::cout << "stmp B" << std::endl;}  
};  
  
int main() {  
    A a;  
    B b;  
    a.stampa();  
    b.stampa();  
}
```


OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {
public:
    A() : a(17)
        { std::cout << "costr A " << a << std::endl; }
    ~A()
        { std::cout << "distr A" << std::endl; }
    void stampa()
        { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
        { std::cout << "costr B " << a << std::endl;}
    ~B()
        { std::cout << "distr B" << std::endl;}
    void stampa()
        { std::cout << "stmp B" << std::endl;}
};

int main() {
    B *b = new B;
    b->stampa();
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- `costr A 17`
`costr B 17`
`stmp B`

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
class B : public A {  
public:  
    B()  
        { std::cout << "costr B " << a << std::endl;}  
    ~B()  
        { std::cout << "distr B" << std::endl;}  
    void stampa()  
        { std::cout << "stmp B" << std::endl;}  
};  
  
int main() {  
    B *b = new B;  
    b->stampa();  
}
```

OOP

- ▶ Cerchiamo di capire quale sia l'ordine di creazione delle classi



```
class A {
public:
    A() : a(17)
    { std::cout << "costr A " << a << std::endl; }
    ~A()
    { std::cout << "distr A" << std::endl; }
    void stampa()
    { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
    { std::cout << "costr B " << a << std::endl;}
    ~B()
    { std::cout << "distr B" << std::endl;}
    void stampa()
    { std::cout << "stmp B" << std::endl;}
};

int main() {
    B *b = new B;
    b->stampa();
    delete b;
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- costr A 17
costr B 17
stmp B
distr B
distr A

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
class B : public A {  
public:  
    B()  
        { std::cout << "costr B " << a << std::endl; }  
    ~B()  
        { std::cout << "distr B" << std::endl; }  
    void stampa()  
        { std::cout << "stmp B" << std::endl; }  
};  
  
int main() {  
    B *b = new B;  
    b->stampa();  
    delete b;  
}
```

OOP

- ▶ Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {
public:
    A() : a(17)
    { std::cout << "costr A " << a << std::endl; }
    ~A()
    { std::cout << "distr A" << std::endl; }
    void stampa()
    { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
    { std::cout << "costr B " << a << std::endl;}
    ~B()
    { std::cout << "distr B" << std::endl;}
    void stampa()
    { std::cout << "stmp B" << std::endl;}
};

int main() {
    A *b = new B;
    b->stampa();
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- `costr A 17`
`costr B 17`
`stmp A`

```
class A {
public:
    A() : a(17)
    { std::cout << "costr A " << a << std::endl; }
    ~A()
    { std::cout << "distr A" << std::endl; }
    void stampa()
    { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
    { std::cout << "costr B " << a << std::endl;}
    ~B()
    { std::cout << "distr B" << std::endl;}
    void stampa()
    { std::cout << "stmp B" << std::endl;}
};

int main() {
    A *b = new B;
    b->stampa();
}
```

OOP

- ▶ Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {
public:
    A() : a(17)
    { std::cout << "costr A " << a << std::endl; }
    ~A()
    { std::cout << "distr A" << std::endl; }
    void stampa()
    { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
    { std::cout << "costr B " << a << std::endl;}
    ~B()
    { std::cout << "distr B" << std::endl;}
    void stampa()
    { std::cout << "stmp B" << std::endl;}
};

int main() {
    A *b = new B;
    b->stampa();
    delete b;
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- `costr A 17`
`costr B 17`
`stmp A`
`distr A`

```
class A {
public:
    A() : a(17)
        { std::cout << "costr A " << a << std::endl; }
    ~A()
        { std::cout << "distr A" << std::endl; }
    void stampa()
        { std::cout << "stmp A" << std::endl; }
    int a;
};

class B : public A {
public:
    B()
        { std::cout << "costr B " << a << std::endl; }
    ~B()
        { std::cout << "distr B" << std::endl; }
    void stampa()
        { std::cout << "stmp B" << std::endl; }
};

int main() {
    A *b = new B;
    b->stampa();
    delete b;
}
```


OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    virtual void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
class B : public A {  
public:  
    B()  
        { std::cout << "costr B " << a << std::endl; }  
    ~B()  
        { std::cout << "distr B" << std::endl; }  
    void stampa()  
        { std::cout << "stmp B" << std::endl; }  
};  
  
int main() {  
    A *b = new B;  
    b->stampa();  
    delete b;  
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- costr A 17
costr B 17
stmp B
distr A

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    ~A()  
        { std::cout << "distr A" << std::endl; }  
    virtual void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
class B : public A {  
public:  
    B()  
        { std::cout << "costr B " << a << std::endl; }  
    ~B()  
        { std::cout << "distr B" << std::endl; }  
    void stampa()  
        { std::cout << "stmp B" << std::endl; }  
};  
  
int main() {  
    A *b = new B;  
    b->stampa();  
    delete b;  
}
```

OOP

- Cerchiamo di capire quale sia l'ordine di creazione delle classi

- costr A 17
costr B 17
stmp B
distr B
distr A

```
class A {  
public:  
    A() : a(17)  
        { std::cout << "costr A " << a << std::endl; }  
    virtual ~A()  
        { std::cout << "distr A" << std::endl; }  
    virtual void stampa()  
        { std::cout << "stmp A" << std::endl; }  
    int a;  
};  
  
class B : public A {  
public:  
    B()  
        { std::cout << "costr B " << a << std::endl; }  
    ~B()  
        { std::cout << "distr B" << std::endl; }  
    void stampa()  
        { std::cout << "stmp B" << std::endl; }  
};  
  
int main() {  
    A *b = new B;  
    b->stampa();  
    delete b;  
}
```

OOP

- Una considerazione finale...

```
#include <iostream>

namespace a {
    int x;
    int y;
    int sommaxy() {
        return x + y;
    }
};

namespace b {
    int h;
    int getH() {
        return h;
    }
    void setH( int hh ){
        h = hh;
    }
};

int main(){
    a::x = 10;
    a::y = 15;
    b::setH( 3120 );
    std::cout << "x + y: " << a::sommaxy()
                << ", h: " << b::getH() << std::endl;
    return 0;
}
```

- Questo programma compila?

OOP

- Una considerazione finale...

```
#include <iostream>

namespace a {
    int x;
    int y;
    int sommaxy() {
        return x + y;
    }
};

namespace b {
    int h;
    int getH() {
        return h;
    }
    void setH( int hh ){
        h = hh;
    }
};
```

```
int main(){
    a::x = 10;
    a::y = 15;
    b::setH( 3120 );
    std::cout << "x + y: " << a::sommaxy()
        << ", h: " << b::getH() << std::endl;
    return 0;
}
```



The screenshot shows a Windows command prompt window with the title bar "C:\windows\...". The window displays the output of the C++ program: "x + y: 25, h: 3120" followed by a prompt "Premere un tasto per continuare . . .". The prompt is followed by a cursor. The window is overlaid on a dark background with some code visible on the left side.

- Cosa sta succedendo?