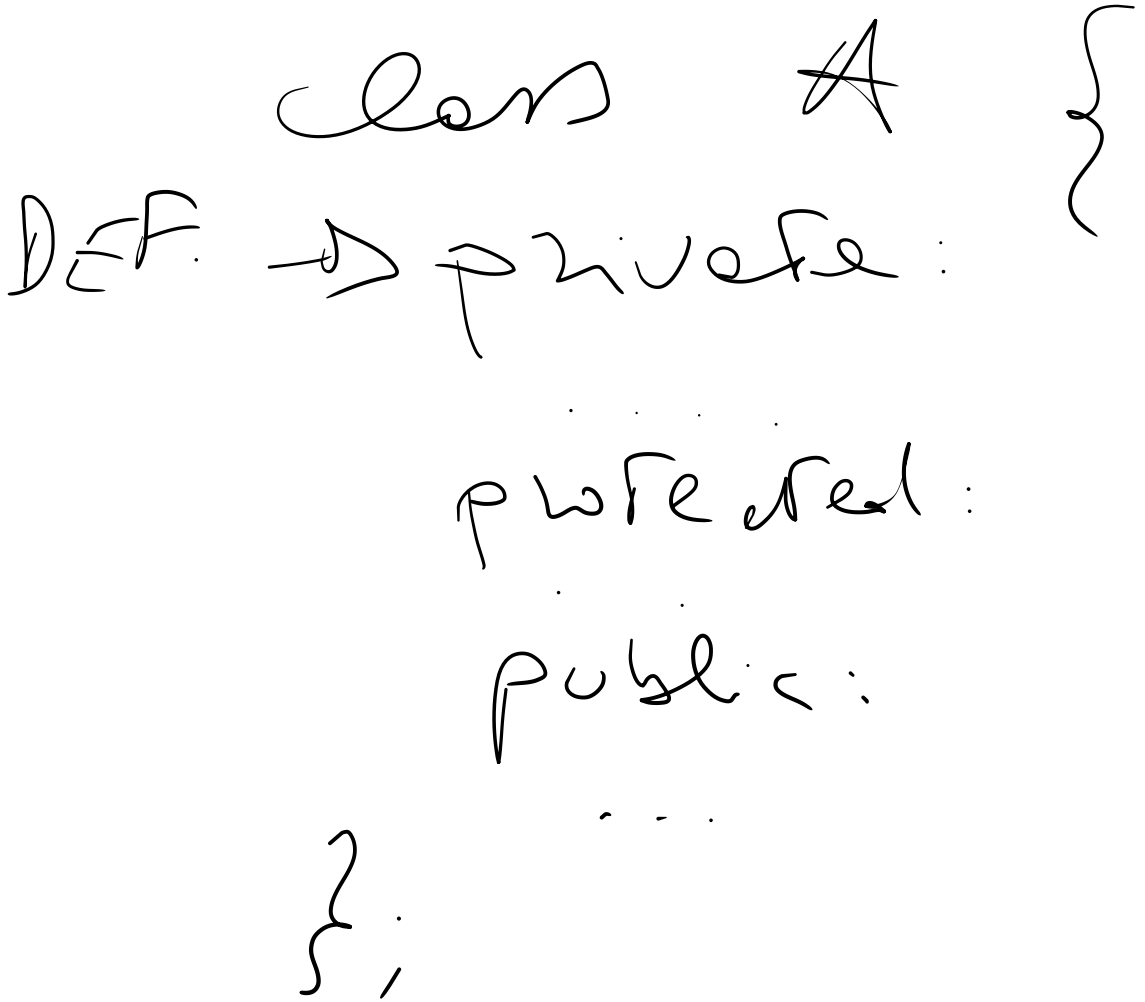
ereditarietà, polimorfismo e template

Modificatori di accesso

- public
- private
- protected



Modificatori di accesso

Tipo di ereditarietà	Base	Derivata
Public	Public	Public
	Protected	Protected
	Private	Inaccessibile
Protected	Public	Protected
	Protected	Protected
	Private	Inaccessibile
private	Public	Private
	Protected	Private
	Private	Inaccessibile

class B : public protected private A

Accedere ai membri della classe base

- Operatore ::

```
class Base {
    public:
        int n;
};
```

```
class Derivata1 : protected Base {
    public:
        Base::i1;
};

class Derivata2 : protected Base {
    ...
};
```

```
int main() {
    Derivata1 d1;
    d1.n = 0; // CORRETTO
    Derivata2 d2;
    d2.n = 0; //ERRORE
}
```

Ereditarietà multipla

- Cosa succede quando istanzio un oggetto di classe C?

```
class A {
    A() { cout << "constructor A" << endl; }
}

class B {
    B() { cout << "constructor B" << endl; }
}
class C : public A, public B{
    C() { cout << "constructor C" << endl; }
}
```

Ereditarietà multipla

- Cosa succede quando se chiamo la funzione foo con c.foo() ?

```
class A {
    A() { cout << "constructor A" << endl; }
    void foo() { cout << "foo B" << endl; }
}

class B {
    B() { cout << "constructor B" << endl; }
    void foo() { cout << "foo B" << endl; }
}
class A : public A, public B{
    C() { cout << "constructor C" << endl; }
}
```

Polimorfismo

- Oggetti di classi diverse rispondono in maniera diversa alla stessa invocazione

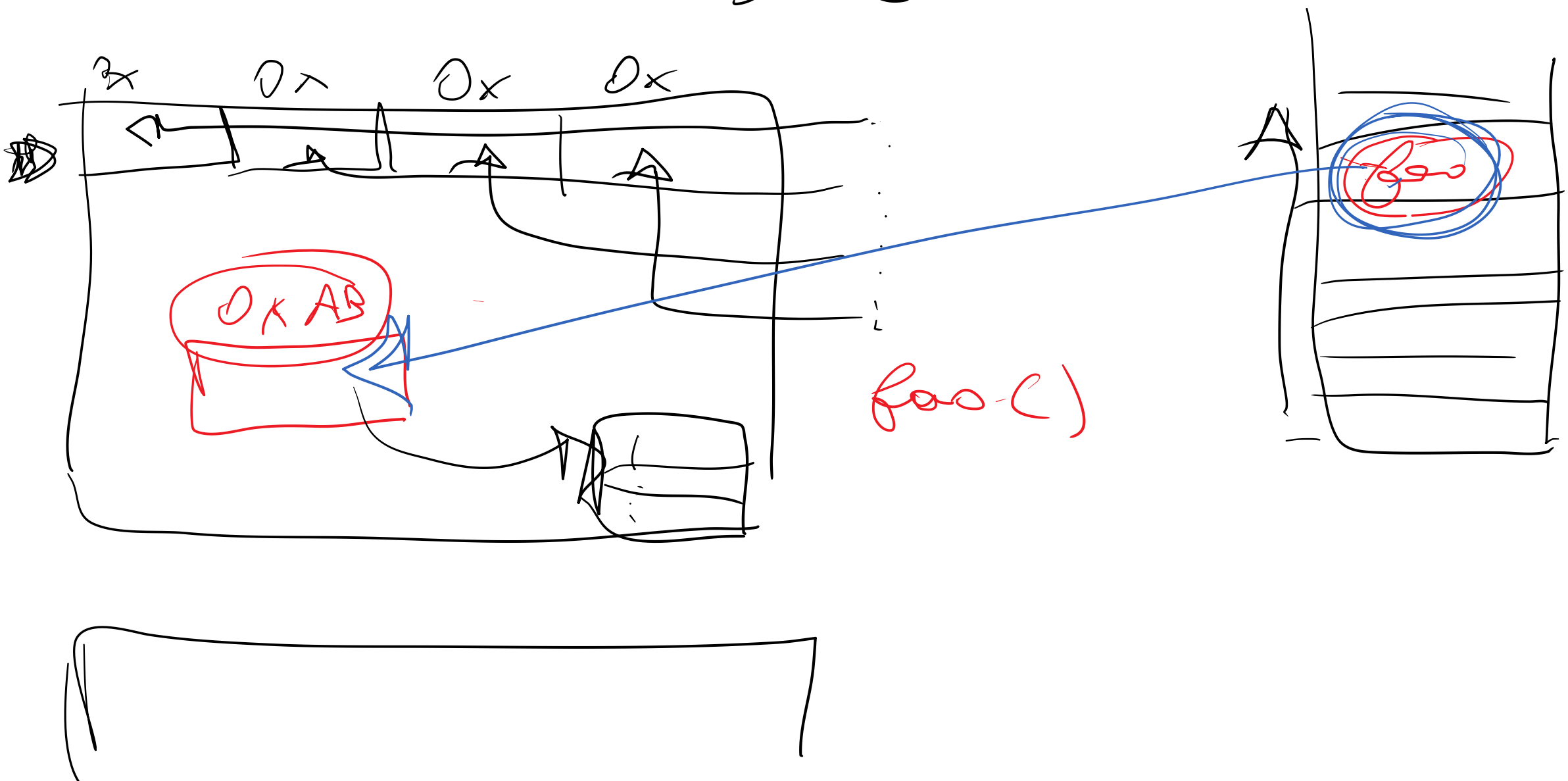
```
A a;
B b;
a.foo();
b.foo();
```

Polimorfismo e binding

- Binding: associazione tra chiamata a una funzione e codice che la implementa
- Stack delle chiamate a funzione
- Implementazione delle funzioni

- Binding statico: risolto a compile-time;
- Binding dinamico: risolto a run-time.

Compilo il Codice

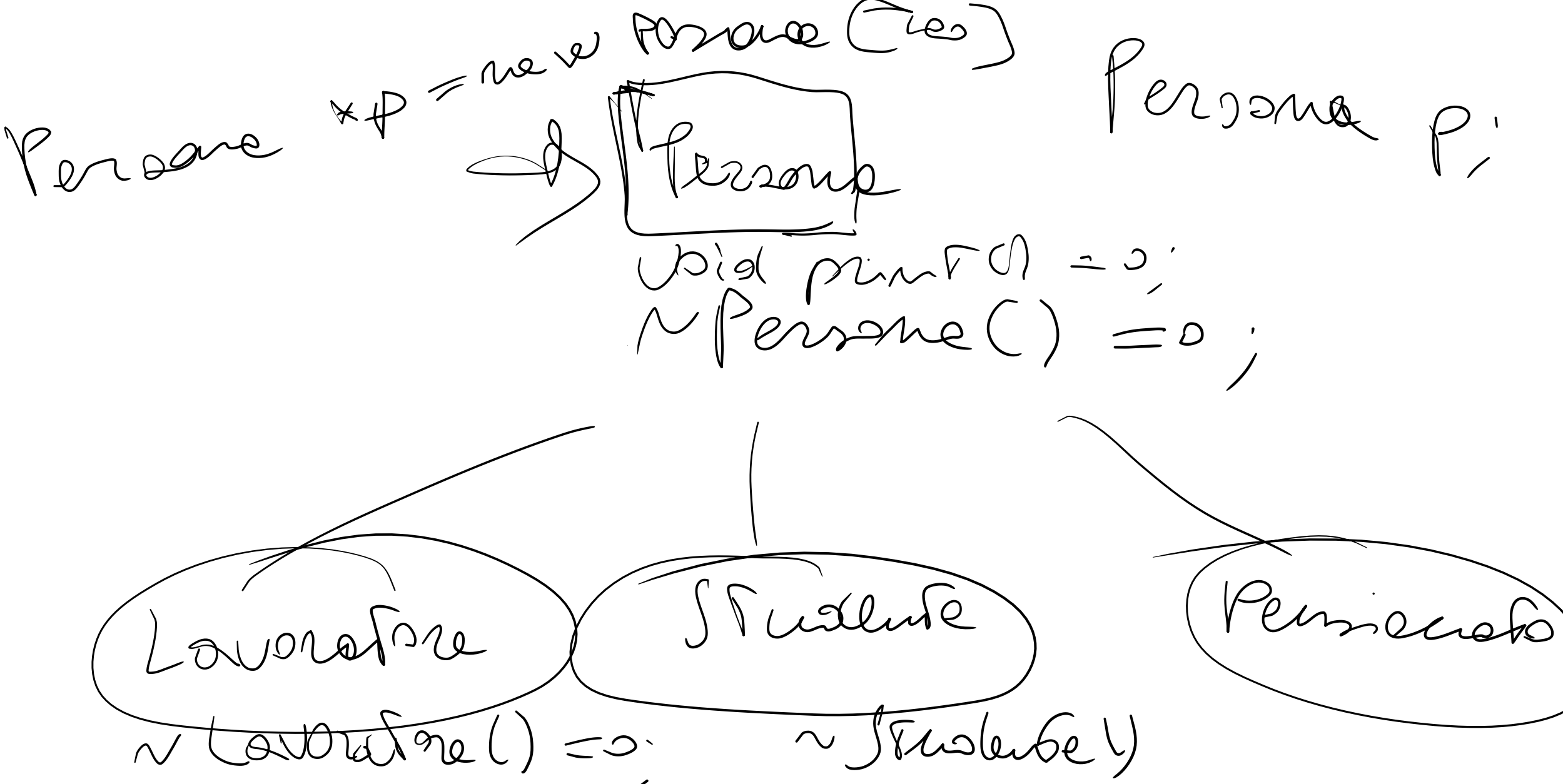


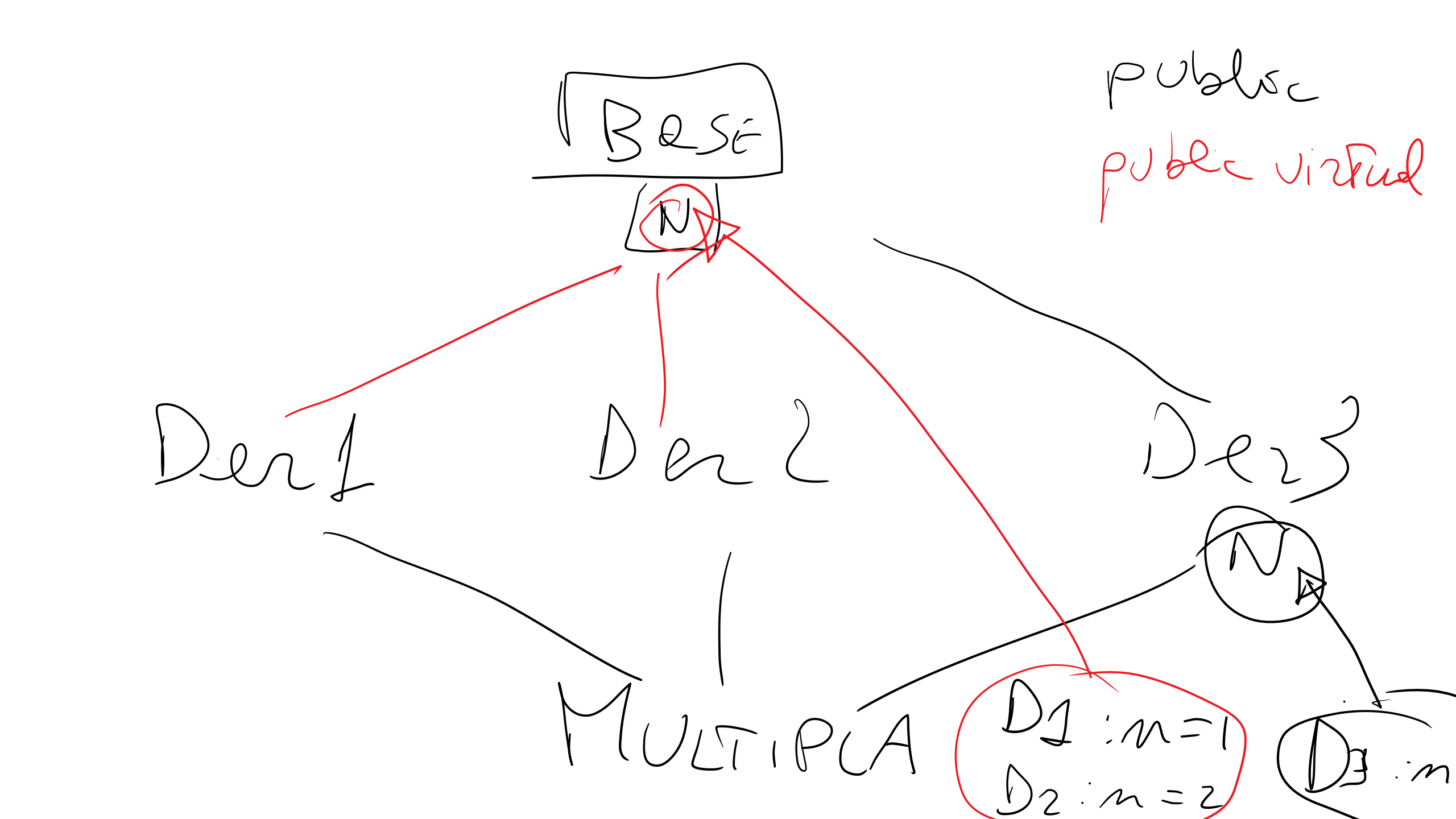
Uso di virtual

- Virtual e riferimenti
- Virtual e puntatori
- Virtual e membri private
- Virtual e ereditarietà multipla
- Pure virtual functions

Persone *p = new Persone(...)

Persone p;




public
public virtual
BASE
N
Der1
Der2
Der3
N
MULTIPLA
D1 : m=1
D2 : m=2
D3 : m

template

- Definiscono un tipo generico di dato

```
template <typename T> return_type function_name(parameter_list)
```

```
template<typename T> void swap(T var1, T var2) {
    cout << "swap " << var1 << " and " << var2 << endl;
    T temp = var1;
    var1 = var2;
    var2 = temp;
}
```

template

- È possibile definire anche più tipi di dato

```
template <class T1, class T2> return_type function_name(parameter_list)
```

template

- Class template

```
template <typename T>
class A {
    public:
    T attributo;
}
```

template

- Uguaglianza tra istanze di classi template
  - Dipende sia dal tipo di dato che dai valori passati

```
template <class tipo_dato, int lunghezza>
class MyArray {
    tipo_dato *array;
    int lunghezza;
    public:
    MyArray() { array = new [lunghezza]; }
}

MyArray<int, 10> a;
MyArray<int, 20> b;
MyArray<double, 10> c;
MyArray<double, 10> d;

a == b --> true
c == d --> true
```

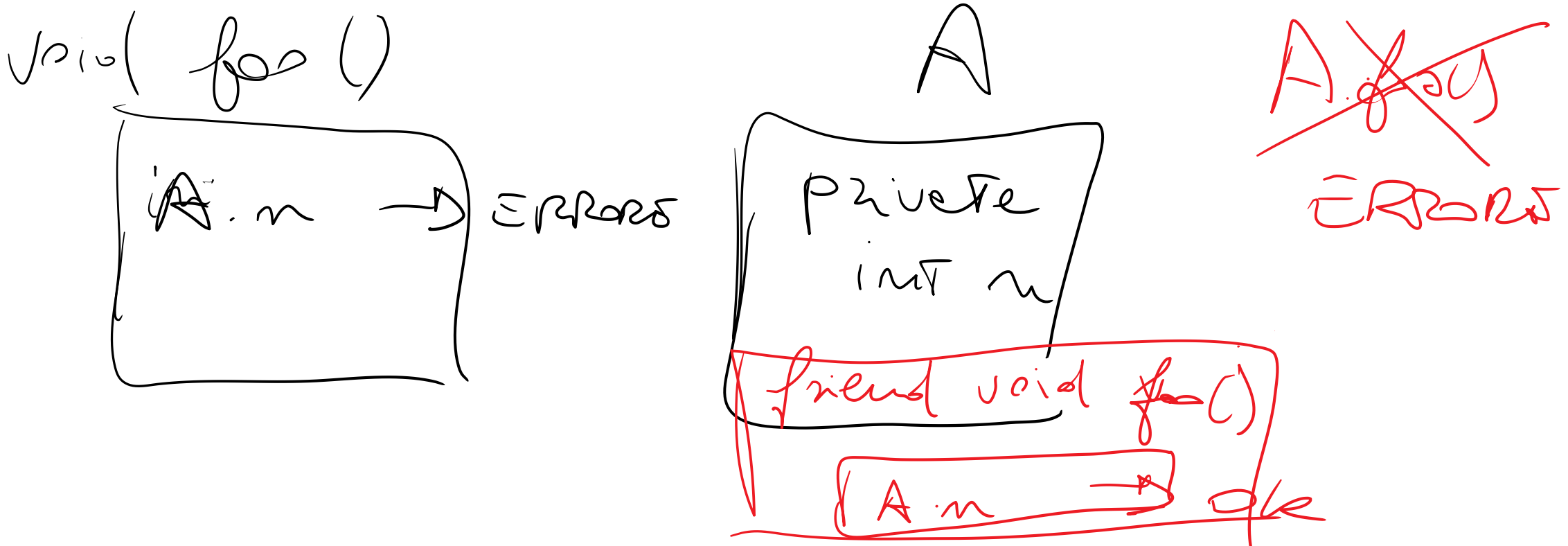
falso

Template - esercizio

- Scrivere una classe template che prenda due tipi di dato generici e li utilizzi come tipi di dato per due attributi della classe stessa. Scrivere metodi set e get e un metodo di stampa, oltre al costruttore.

friend

- La keyword friend fornisce ad una funzione o ad una classe accesso ai membri privati e protetti della classe in cui appare



friend - utilizzi

1. Designare funzioni o membri di altre classi come friend della classe

Esempio: operatore <<

friend - utilizzi

1. Designare funzioni o membri di altre classi come friend della classe
2. Definire funzioni non-membro all'interno delle classi friend

```
class A {
    private:

    int n = 10;
    friend void set_member(A& a, int val) {
        a.n = val;
    }

    public:
    void set_n(int val) {
        n = val;
    }
};
```

friend - utilizzi

1. Designare funzioni o membri di altre classi come friend della classe
2. Definire funzioni non-membro all'interno delle classi friend
3. Designare classi che verranno utilizzate successivamente come friend della classe stessa:
   - Elaborated-class-specifier – la classe può anche non essere stata ancora definita;

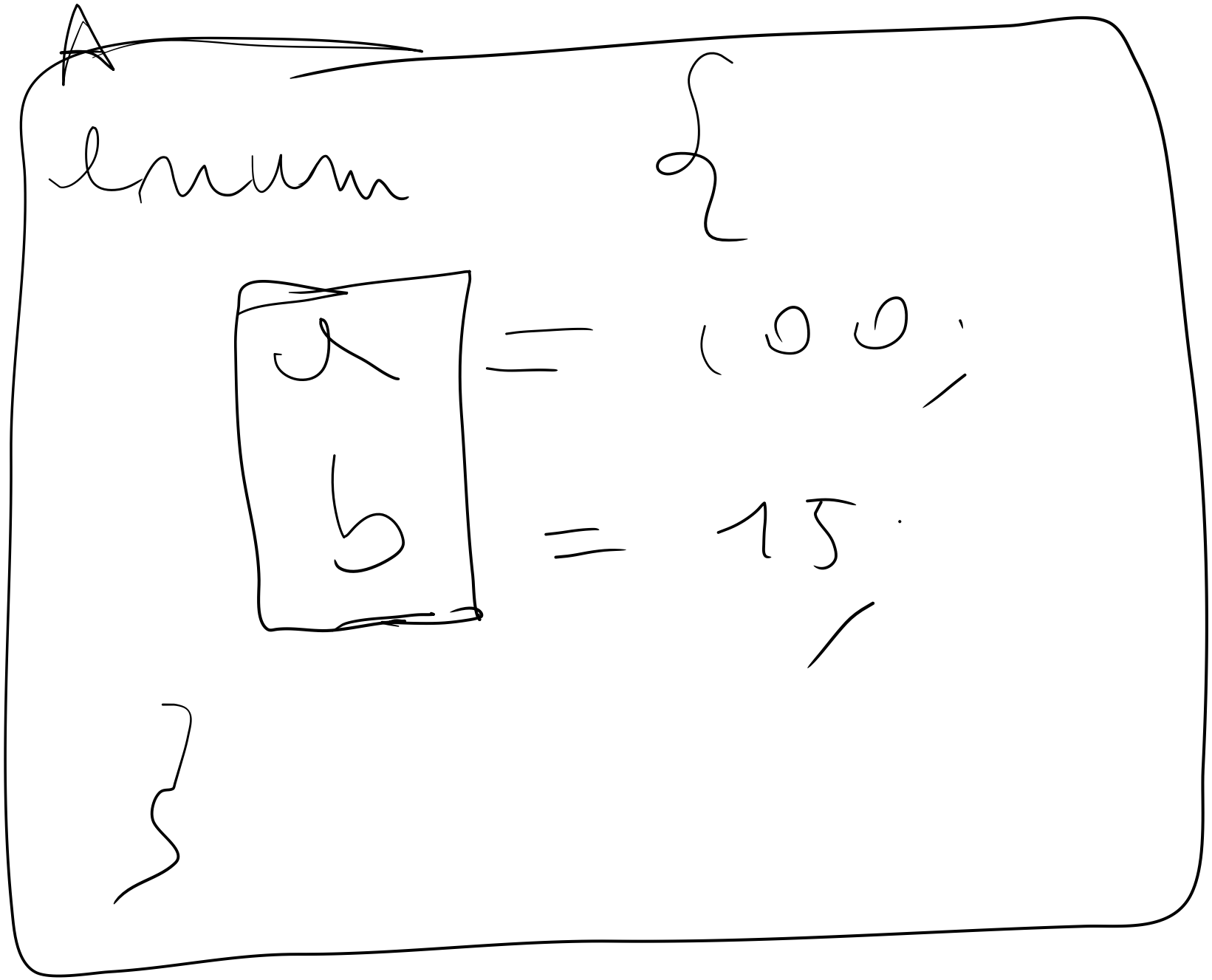
     friend class X;
   - Simple-type-specifier – la friendship viene effettuata solo se il tipo di dato dichiarato come friend è una classe, struct o union, altrimenti viene ignorato. La classe deve essere già dichiarata.

     friend Y;

A è friend di X

A può accedere ai private di X

X non può accedere



A : a

A : b