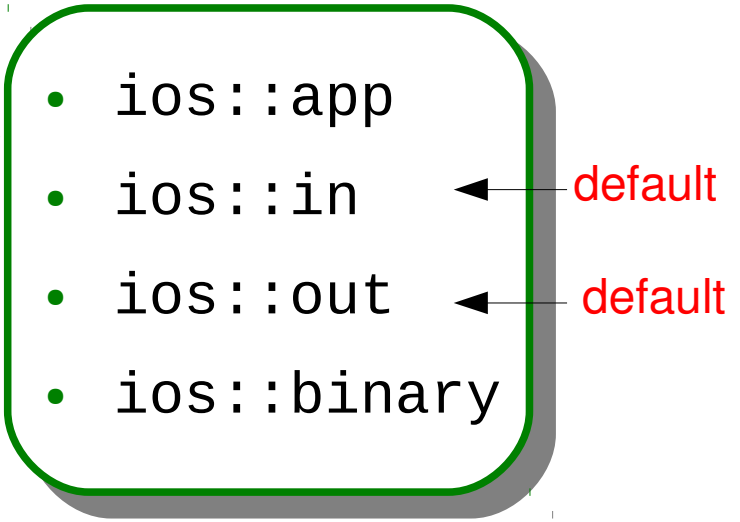
Lettura e scrittura
di file di dati input/output

Lettura e scrittura da disco

- Molto semplice in C++: si fa esattamente come se fosse una tastiera (se sto leggendo da disco) o lo schermo (se sto scrivendo su disco)
- Bisogna aggiungere `#include <fstream>` oltre al solito `iostream`

```
#include <iostream>
#include <fstream>


int main()
{
  int a,b;

  std::ofstream outFile ("outFile.txt", std::ios::out);
  outFile << "12 24" << std::endl;
  outFile.close();

  std::ifstream inFile;
  inFile.open("outFile.txt", std::ios::in);
  inFile >> a >>b ;
  inFile.close();

  std::cout << "Ho letto i numeri " << a << " e " << b << std::endl;
}
```



Se un file ha molte righe...

```
int num;
std::ifstream inFileLong;
inFileLong.open ("list.txt", std::ios::in);
while (!inFileLong.eof())
{
    inFileLong >> num;
    std::cout << "Ho letto il numero " << num << std::endl;
}
inFileLong.close ();
```

- Il metodo `eof ()` restituisce un valore `true` se la fine del file è stata raggiunta

```
char line[256];
inFileLong.open ("listName.txt", std::ios::in);
while (!inFileLong.eof())
{
    inFileLong.getline (line, 256);
    std::cout << "Ho letto la riga " << line << std::endl;
}
inFileLong.close ();
```

- Il metodo `getline (char* line, int num)` estrae `num` caratteri, riga per riga, e li salva nel vettore di caratteri `line`

Elenco del telefono

- Provate a scrivere un programma che legga un file di testo con
  - elenco di nomi di persone
  - numero di telefono delle persone

- Salvate queste informazioni in una mappa stl del tipo `std::map<std::string, std::string>`

- Implementate una funzione che interroghi l'elenco telefonico!

Ereditarietà

Riassunto

- **Programmazione C:** utilizzare il C++ come un linguaggio procedurale, cioè come se fosse C
- **Programmazione *object-based*:** incapsulare i programmi all'interno di classi, che nascondono le funzionalità, sistematizzano e semplificano il codice
- **Programmazione *template* (STL):** generalizzare le applicazioni a tutti i tipi che condividono le stesse funzionalità e sfruttare potenti librerie già esistenti
- **Programmazione *object-oriented*:** introdurre relazioni gerarchiche fra tipi, attraverso il meccanismo dell'ereditarietà

Programmazione *object-oriented*

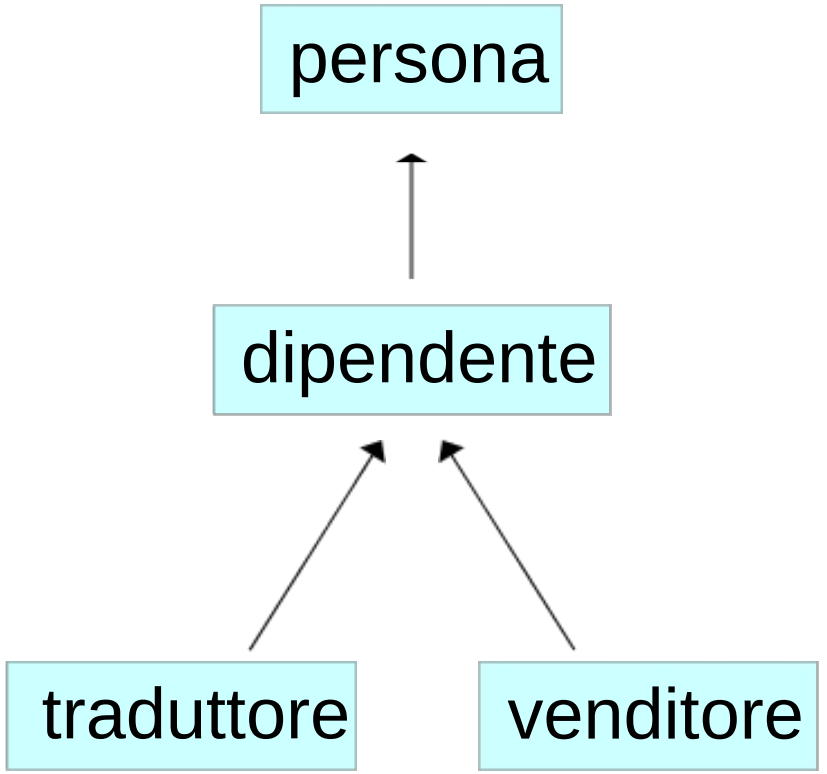
- **Riutilizzare funzioni già scritte**:
  piuttosto che re-implementare caratteristiche condivise da due classi, si fa in modo che le classi le condividano

- **Stabilire il comportamento di una classe** (l'*interfaccia*) in modo disgiunto dalla sua implementazione

- Le classi che hanno già implementate le funzioni o le interfacce, sono **classi "base"**, quelle che ereditano le funzioni o le interfacce sono **classi "derivate"**

Un esempio: sedie

Un albero di ereditarietà



- la **classe base** contiene le proprietà fondamentali dell'oggetto
- una **classe derivata** è una **specializzazione** della classe base, e ne mantiene la natura
- si possono costruire **specializzazioni di specializzazioni** senza limiti
- sono tutte classi "persona"!

La classe base: persona

```
class persona
{
  // accessibili a tutti
  public:

    persona (std::string nome, int ADN, int MDN, int GDN) ;
    virtual ~persona () ;

    //! ritorna la data di nascita a schermo
    virtual void stampaNascita () ;
    //! scrive il nome a schermo
    virtual void stampaNome () ;

  // accessibili a tutte le classi derivate
  protected:

    std::string m_nome ;

  // accessibili a questa classe, non a quelle derivate
  private:

    int m_annoDiNascita ;
    int m_meseDiNascita ;
    int m_giornoDiNascita ;

} ;
```

- **la parola chiave virtual** introduce le funzioni reimplementabili dalle classi derivate
- **necessario al distruttore**
- **il campo public** è accessibile anche alle classi derivate
- **il campo protected** è accessibile anche e soltanto alle classi derivate
- **il campo private** è accessibile solo alla classe base

Classe derivata: dipendente

```
#include "persona.h"

class dipendente : public persona
{
  // accessibili a tutti
  public:

    dipendente (std::string qualifica, std::string nome,
                int ADN, int MDN, int GDN) ;
    virtual ~dipendente () = 0 ;

    //! scrive il nome a schermo
    virtual void stampaNome () ;

  // accessibili a tutte le classi derivate
  protected:

    std::string m_qualifica ;

  // accessibili a questa classe, non a quelle derivate
  private:

} ;
```

- : public persona
  dice che **la classe dipendente eredita da persona**
- c'è una **variabile in più** rispetto a persona
- **il metodo stampaNome** è presente perché viene reimplementato aggiungendo la qualifica
- c'è un metodo (il distruttore) che è dichiarato = 0: **la classe è <u>puramente virtuale</u>**. Significa che può soltanto fare da "base" per altre classi (non si può creare un oggetto "dipendente")

Classe derivata: venditore

```
#include "dipendente.h"

class venditore : public dipendente
{
  // accessibili a tutti
  public:

    venditore (double portafoglio, std::string nome,
               int ADN, int MDN, int GDN) ;
    virtual ~venditore () ;

    //! scrive il nome a schermo
    virtual void stampaNome () ;

  // accessibili a tutte le classi derivate
  protected:

  // accessibili a questa classe, non a quelle derivate
  private:

    double m_portafoglio ;

} ;
```

- di nuovo **stampaNome è reimplementato**
- **c'è una nuova variabile** private (perché non deve essere letta da nessun altro)
- la classe venditore **eredita esplicitamente solo da dipendente**, NON da persona, ma è di tipo persona

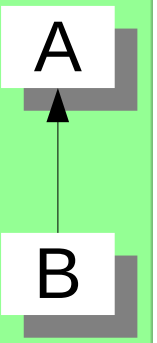
Accortezze

- **il distruttore** di una classe base sia sempre virtual
- **non usare gli stessi nomi** per cose diverse,
  se non sono la stessa funzione, altrimenti ne esce un pasticcio
- **separare le varie classi**, ciascuna con i relativi .h e .cc

Essere o avere?

- quando si utilizza l'ereditarietà?

- **essere:** un oggetto B è di tipo A, cioè tutto quello che può fare A lo può fare B, ma non viceversa.
  In questo caso B eredita da A (A è classe base, B è classe derivata)



- **avere:** un oggetto A contiene fra i propri elementi un oggetto B.
  In questo caso NON si utilizza ereditarietà, ma B diventa un membro di A

- **altro:** due oggetti non devono necessariamente essere in relazione di ereditarietà

Implementazione: stampaNome

```
void persona::stampaNome ()
{
  std::cout << m_nome
            << std::endl ;
  return ;

}
```

```
void dipendente::stampaNome ()
{
  std::cout << m_qualifica << " "
            << m_nome
            << std::endl ;
  return ;

}
```

- nel .h **la funzione è dichiarata virtual,** nel .cc non è necessario
- la funzione è **implementata sia nella classe base che in quella derivata**
- l'implementazione della classe derivata, se c'è, **sostituisce quella della classe base**

Implementazione: ctor e dtor

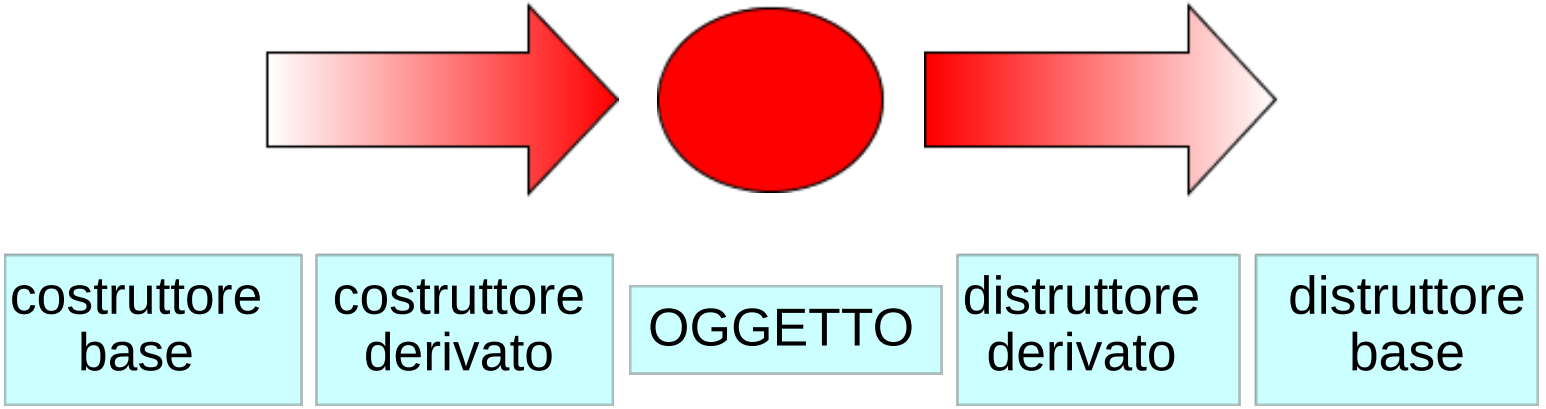
- **il costruttore della classe base** non ha nulla di speciale

- il costruttore della classe derivata chiama quello della classe **da cui deriva direttamente** (nella **lista di inizializzazione**)

```
dipendente::dipendente (std::string qualifica, std::string nome,
                        int ADN, int MDN, int GDN) :
  persona (nome, ADN, MDN, GDN),
  m_qualifica (qualifica)
{
  std::cout << m_nome
            << " viene assunto con la qualifica di: "
            << m_qualifica
            << std::endl ;
}
```

Che cosa succede

- **l'ordine di costruzione** e distruzione va dalla base alla derivata e torna indietro



- **<u>non</u> utilizzare mai funzioni virtual** nei ctor e dtor, perchè non è ovvio quale implementazione venga chiamata

Copy ctor e operator=

- **se non sono definiti esplicitamente**,
  vengono eseguite correttamente le copie dei membri della classe che hanno i loro copy ctor ed operator= definiti

- **se sono definiti esplicitamente**,
  devono richiamare esplicitamente al proprio interno
  i rispettivi operatori della classe base

il copy ctor esplicitamente

```
persona::persona (const persona & original) :
  m_nome (original.m_nome) ,
  m_annoDiNascita (original.m_annoDiNascita) ,
  m_meseDiNascita (original.m_meseDiNascita) ,
  m_giornoDiNascita (original.m_giornoDiNascita)
{
  std::cout << m_nome
            << " nasce il "
            << m_giornoDiNascita
            << "/" << m_meseDiNascita
            << "/" << m_annoDiNascita
            << " (copy ctor)" << std::endl ;
}
```

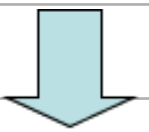
```
dipendente::dipendente (const dipendente & original) :
   persona (original) ,
   m_qualifica (original.m_qualifica)
 {
   std::cout << m_nome
             << " viene assunto con la qualifica di: "
             << m_qualifica
             << " (copy ctor)" << std::endl ;
 }
```

```
traduttore::traduttore (const traduttore & original) :
   dipendente (original) ,
   m_lingua (original.m_lingua)
 {
   std::cout << " -> specializzazione in: "
             << m_lingua
             << " (copy ctor)" << std::endl ;
 }
```

L'operator= esplicitamente

```
persona &
persona::operator= (const persona & original)
  {
    if (&original == this) return *this ;
    m_nome = original.m_nome ;
    m_annoDiNascita = original.m_annoDiNascita ;
    m_meseDiNascita = original.m_meseDiNascita ;
    m_giornoDiNascita = original.m_giornoDiNascita ;
    std::cout << m_nome
              << " nasce il "
              << m_giornoDiNascita
              << "/" << m_meseDiNascita
              << "/" << m_annoDiNascita
              << " (operator=)" << std::endl ;
    return *this ;
  }
```

```
dipendente &
dipendente::operator= (const dipendente & original)
{
  if (&original == this) return *this ;

  this->persona::operator= (original) ;
  m_qualifica = original.m_qualifica ;
  std::cout << m_nome
            << " viene assunto con la qualifica di: "
            << m_qualifica
            << " (operator=)" << std::endl ;

  return *this ;
}
```



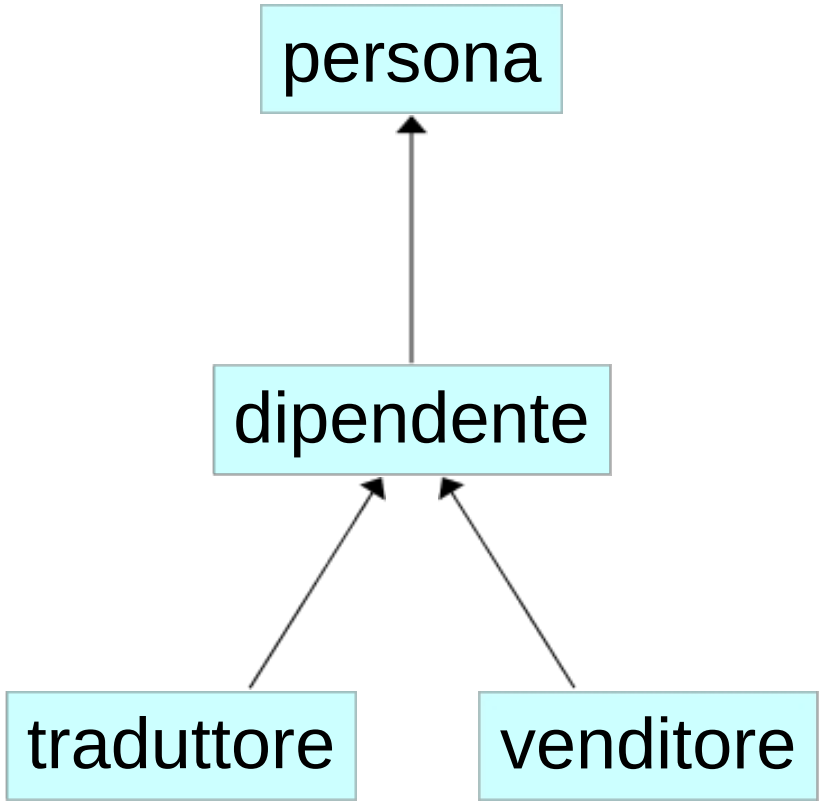
```
traduttore &
traduttore::operator= (const traduttore & original)
{
  if (&original == this) return *this ;

  this->dipendente::operator= (original) ;
  m_lingua = original.m_lingua ;
  std::cout << " -> specializzazione in: "
            << m_lingua
            << " (operator=)" << std::endl ;

  return *this ;
}
```

Il polimorfismo

- **il polimorfismo**:
  ogni puntatore (o referenza) di una classe base è in grado di indicizzare una qualunque delle classi derivate dalla base

- *la natura dell'oggetto indicizzato (di quale classe derivata di tratti) viene deciso durante l'esecuzione del programma (al runtime), grazie al* **dynamic binding**

Un esempio



- un **vettore di puntatori a persona** è in grado di indicizzare puntatori a persona, traduttore, venditore
- si creano gli oggetti
- si riempie un vettore con puntatori ad essi
- si verifica se ogni puntatore viene riconosciuto correttamente

L'implementazione

```
// creo tre persone (diverse)
persona gianni ("gianni",2006,2,29) ;
traduttore luigi ("milanese","luigi",2006,2,25) ;
venditore susanna (10000,"susanna",2006,3,15) ;

// creo un vettore di puntatori a persone (alla classe base!)
std::vector<persona *> gente ;

// riempio il vettore di puntatori alla classe base con le varie persone
// (che possono anche appartenere a classi derivate)
gente.push_back (&gianni) ;
gente.push_back (&luigi) ;
gente.push_back (&susanna) ;

std::cout << "\nINIZIA L'ITERAZIONE\n\n" ;
// itero sul vettore e faccio stampare il nome delle persone
for (std::vector<persona *>::iterator genteIt = gente.begin () ;
     genteIt != gente.end () ;
     ++genteIt)
  {
    (*genteIt)->stampaNome () ;
  } // chiusa l'iterazione sul vettore

std::cout << "\nFINISCE L'ITERAZIONE\n\n" ;
```

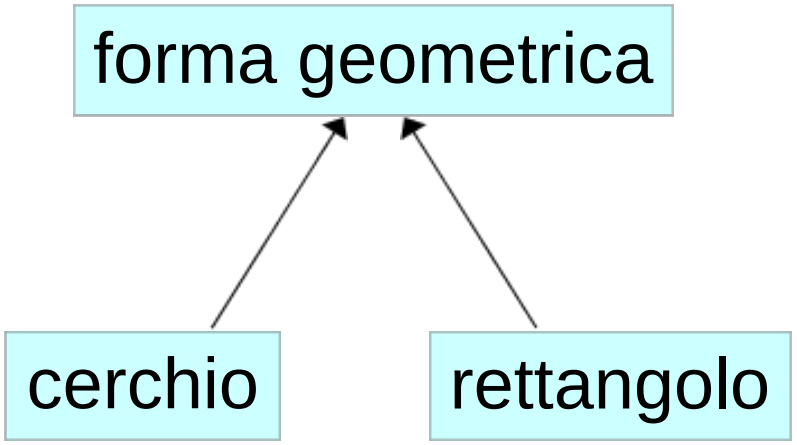
Il risultato

```
INIZIA L'ITERAZIONE

gianni
traduttore luigi
 -> specializzazione in: milanese
venditore susanna
 -> con portafoglio di: 10000 euro

FINISCE L'ITERAZIONE
```

- gianni è di tipo persona
- luigi è di tipo traduttore

- susanna è di tipo venditore

Un altro esempio



- la **forma geometrica** è una classe base virtuale che contiene l'interfaccia a metodi comuni (area, perimetro, disegnami)
- le **singole forme** geometriche implementano questi comportamenti a seconda di che cosa sono

I quadrati NON sono rettangoli

- è giusto scrivere:
  class quadrato: public rettangolo?
- viene rispettata la relazione di “essere”?
- **NO!** perchè un rettangolo può cambiare la lunghezza di uno dei suoi lati, un quadrato no

- **essere:** un oggetto B è di tipo A, cioè tutto quello che può fare A lo può fare B, ma non viceversa.
  In questo caso B eredita da A (A è classe base, B è classe derivata)

Classi puramente virtuali

```
#include "persona.h"

class dipendente : public persona
{
  // accessibili a tutti
  public:

    dipendente (std::string qualifica, std::string nome,
                int ADN, int MDN, int GDN) ;
    virtual ~dipendente () = 0 ;

    //! scrive il nome a schermo
    virtual void stampaNome () ;

  // accessibili a tutte le classi derivate
  protected:

    std::string m_qualifica ;

  // accessibili a questa classe, non a quelle derivate
  private:

} ;
```

- un metodo della classe è **definito puramente virtuale** con “= 0”
- la **scelta** di quale metodo non è fondamentale
- **non si può definire un oggetto di tipo puramente virtuale**
- la classe **serve come base** per altre classi derivate
- la classe derivata **deve implementare il metodo puramente virtuale**

Un caso particolare: le interfacce

- **definire il comportamento** di un insieme di classi con una classe base puramente virtuale
- le **classi derivate** ereditano dalla classe base e implementano gli algoritmi effettivamente
- grazie al polimorfismo, i **programmi possono essere scritti con la classe base** e poi utilizzati con quella derivata

L'amicizia

- il meccanismo friend permette ad una classe di
  garantire l'accesso ai suoi membri non pubblici
  a funzioni o classi

- si realizza con la **parola chiave friend**

- è una **relazione più stretta dell'ereditarietà**, perché ogni funzione dichiarata friend di una classe vede i membri privati di tutti gli oggetti della classe di cui è amica

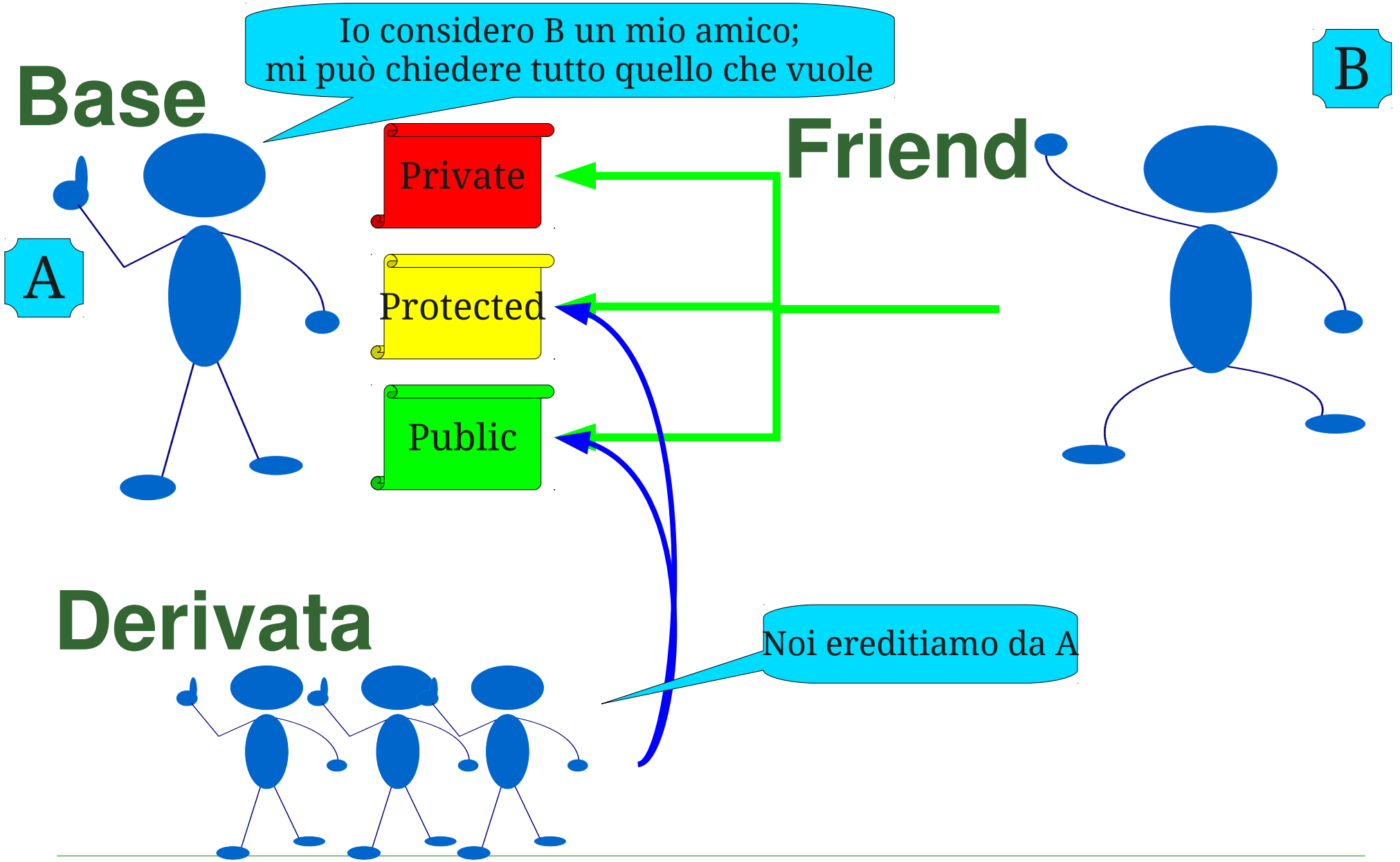
```
class A{
...
friend class B;
friend
somma(int,int);
...
};
```

```
class B {
...
};

double somma ( a, b)
```

Esercizi

- scrivere una semplice struttura di classi (ad esempio le persone)
- in particolare:
  - implementare i tipi di costruttore e l'operator= con un output in ciascuno per verificare l'ordine di chiamata
  - implementare i distruttori con output
  - implementare una funzione virtuale nella classe base implementata nelle classi derivate (con output)
  - implementare una funzione virtuale nella classe base non implementata nelle classi derivate (con output)
  - fare un test del polimorfismo

Comics


Io considero B un mio amico;
mi può chiedere tutto quello che vuole
Base
Friend
B
A
Private
Protected
Public
Derivata
Noi ereditiamo da A