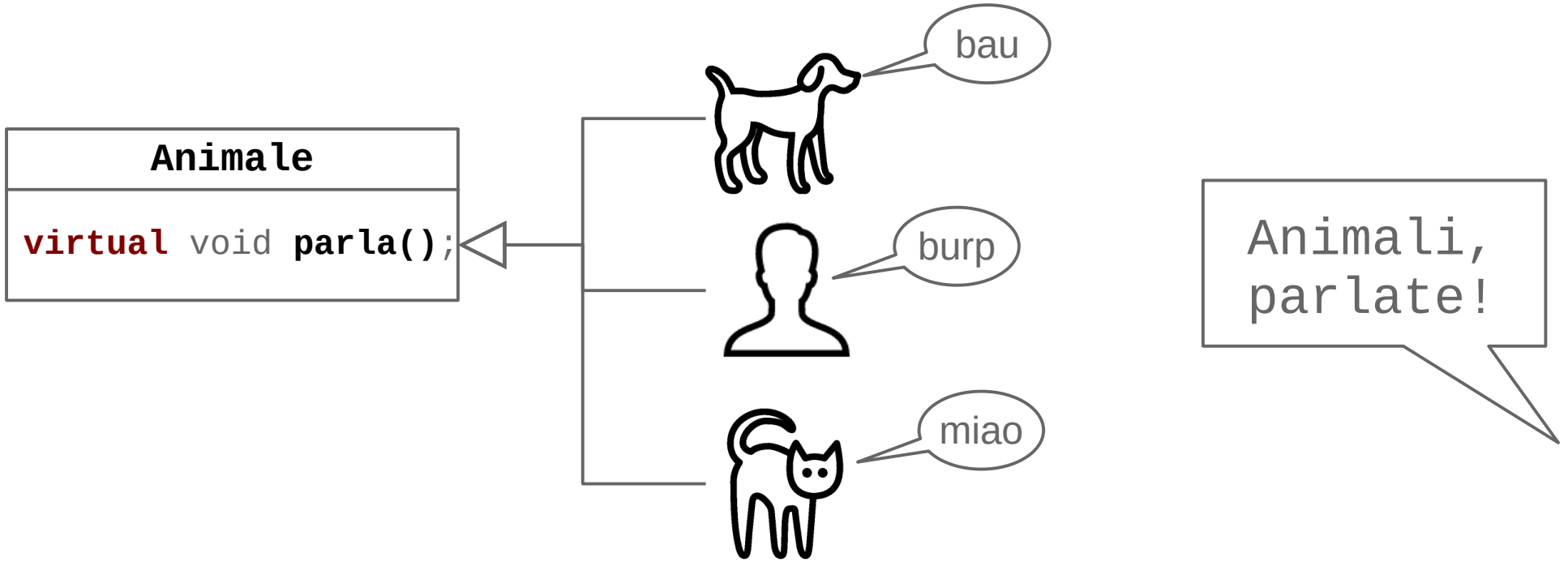
Metodi informatici della Fisica



C++ 11: polimorfismo runtime

Susanna.Costanza@unipv.it
Andrea.Negri@unipv.it

Paradigmi OO

- **Classe**
  - Definizione di un nuovo tipo, unione di tipi già noti
  - Unificazione tra rappresentazione dei dati e metodi per la loro manipolazione
- **Incapsulamento**
  - Lo stato dell'oggetto (la rappresentazione) non è “esposto” all'utente, che ha accesso ad un'interfaccia pubblica decisa dalla sviluppatore
- **Ereditarietà**
  - Possibilità di derivare nuove classi da quelle già esistenti. Le classi derivate ereditano caratteristiche e proprietà delle classi base
- **Polimorfismo**
  - Possibilità di definire un singola interfaccia (per es il nome di un metodo) che viene mappata su molteplici definizioni in base al tipo dei parametri in gioco
  - Tre tipologie: overloading, template e **polimorfismo run-time**

Premessa: puntatori e classi

- Le classi definiscono un nuovo tipo
  - quindi possono esistere puntatori a oggetti di tale tipo
- Esempio con classe std::string
  - è possibile definire un puntatore di tipo `std::string` ed assegnargli l'indirizzo di un oggetto `std::string`

```
std::string s("Boh"); // Oggetto std::string
std::string* p;       // Puntatore a std::string
p=&s;                 // Assegnazione puntatore
```

  - Oppure usare un puntatore a `std::string` per accedere ad un oggetto della classe `std::string` creato dinamicamente con l'operatore `new`

```
std::string *q = new std::string("a caso");
```

Premessa: puntatori e classi

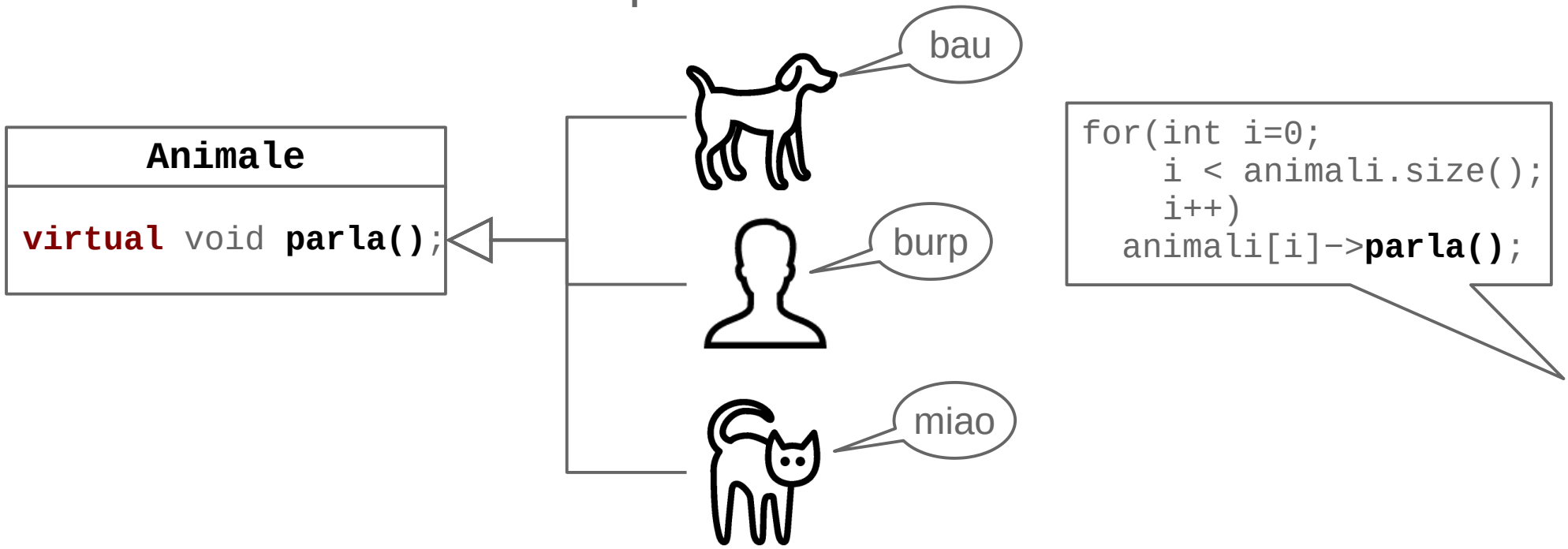
- Chiamata dei metodi
  - Sappiamo che per chiamare i metodi di un oggetto di una classe si utilizza l'operatore punto: "**.**"
  - Ma se abbiamo un puntatore ad un oggetto di una classe occorre utilizzare un nuovo operatore: "**->**"
  - Ovviamente il compilatore da errore se "**.**" è utilizzato con un puntatore, o vice versa
- Esempio

```
std::string  s("ciccio"); // Oggetto std::string
std::string* p = &s;      // Puntatore a std::string

s.size();  // Chiamata del metodo tramite oggetto
p->size(); // Chiamata del metodo tramite puntatore
```

puntatori+classi.cxx

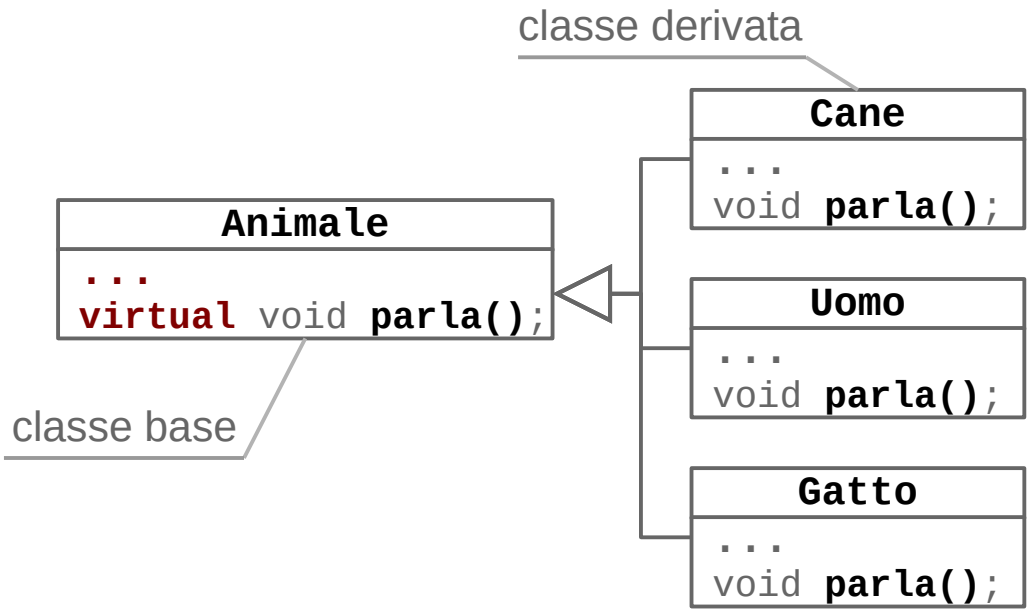
Polimorfismo run-time

- E' un tipo di polimorfismo che avviene **durante** l'esecuzione del programma
  - Cioè non avviene al momento della compilazione, come l'overloading delle funzioni o l'utilizzo dei template
  - E' basato su **puntatori** ed **ereditarietà**
  - Non è banale da capire



Polimorfismo run-time

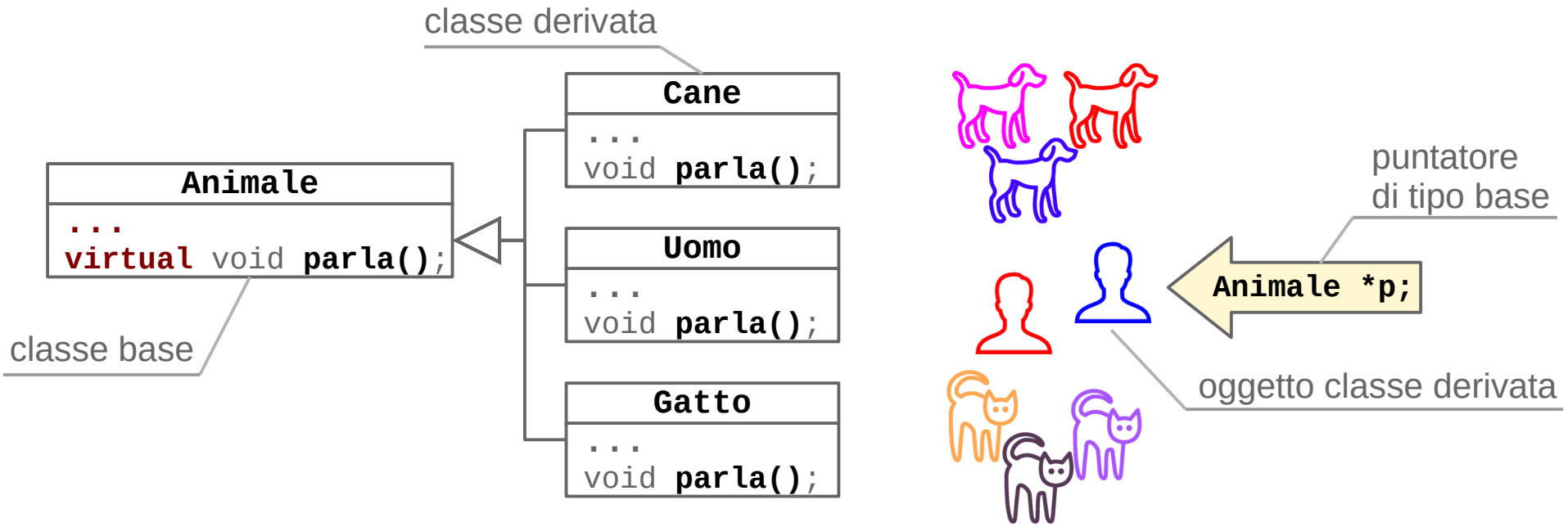
- Come si prepara:
  - nella classe base si etichetta uno o più metodi con il qualificatore **virtual**
  - cioè si dichiara che quei metodi sono virtualizzabili
  - le classi derivate fanno l'**overloading** di tali metodi



- **NB**: è possibile fare l'overloading anche se i metodi base non sono dichiarati virtual
  - Non è questa la peculiarità di virtual

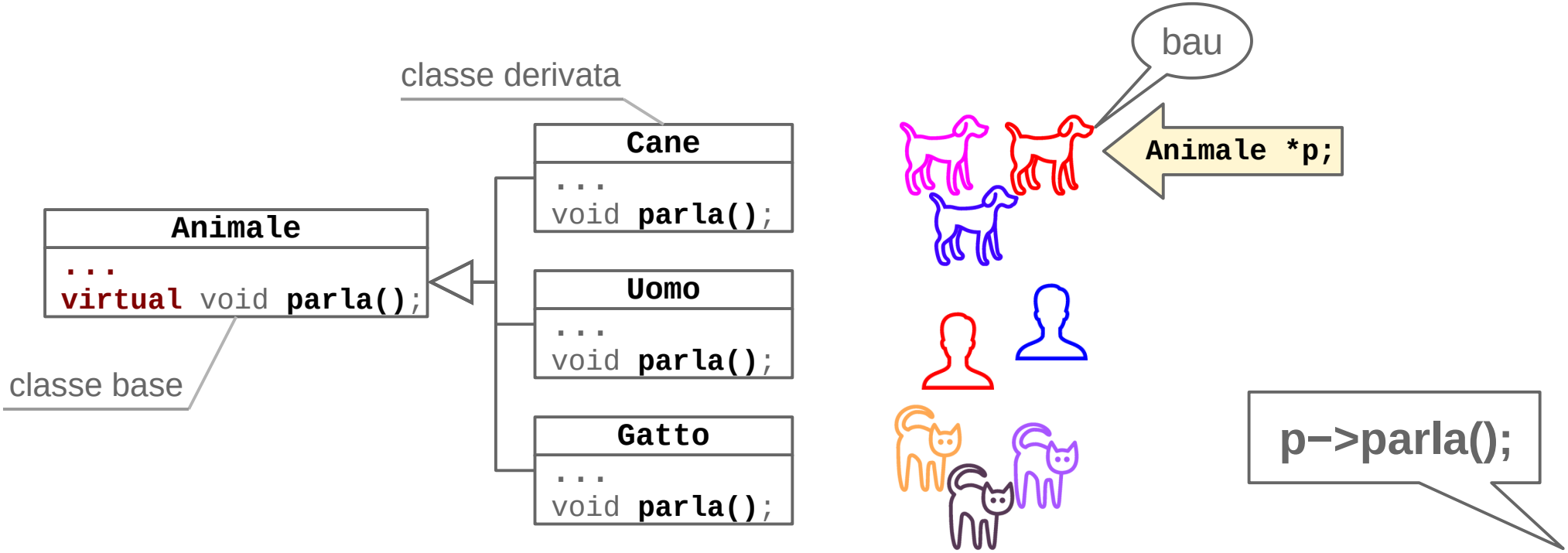
Polimorfismo run-time

- Come si usa:
  - Nel codice si definiscono **puntatori** alla classe **base**
  - E con essi si **puntano** oggetti della classe **derivata**
  - Cioè si usano puntatori alla classe base per puntare ad oggetti della classe derivata



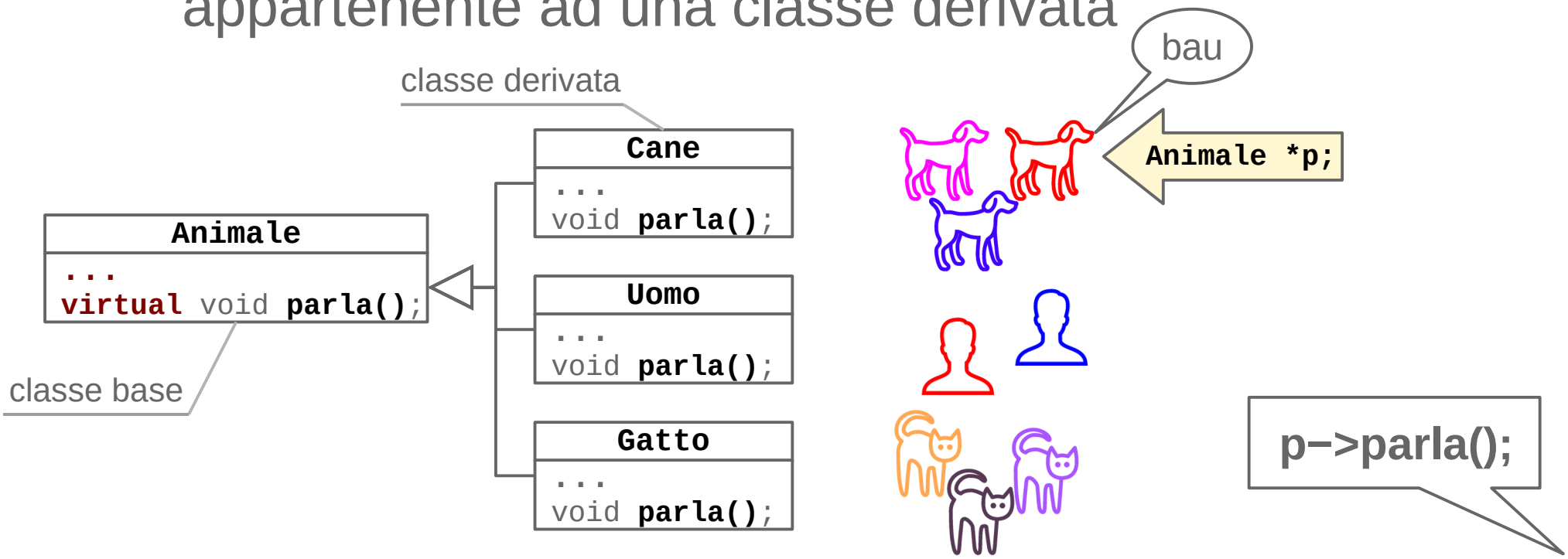
Polimorfismo run-time

- Quando, tramite un puntatore alla classe **base**, viene chiamato un metodo definito **virtual**
  - il sistema riconosce **run-time** il tipo dell'oggetto puntato, cioè a quale classe **derivata** appartiene
  - ed esegue la versione del metodo corrispondente



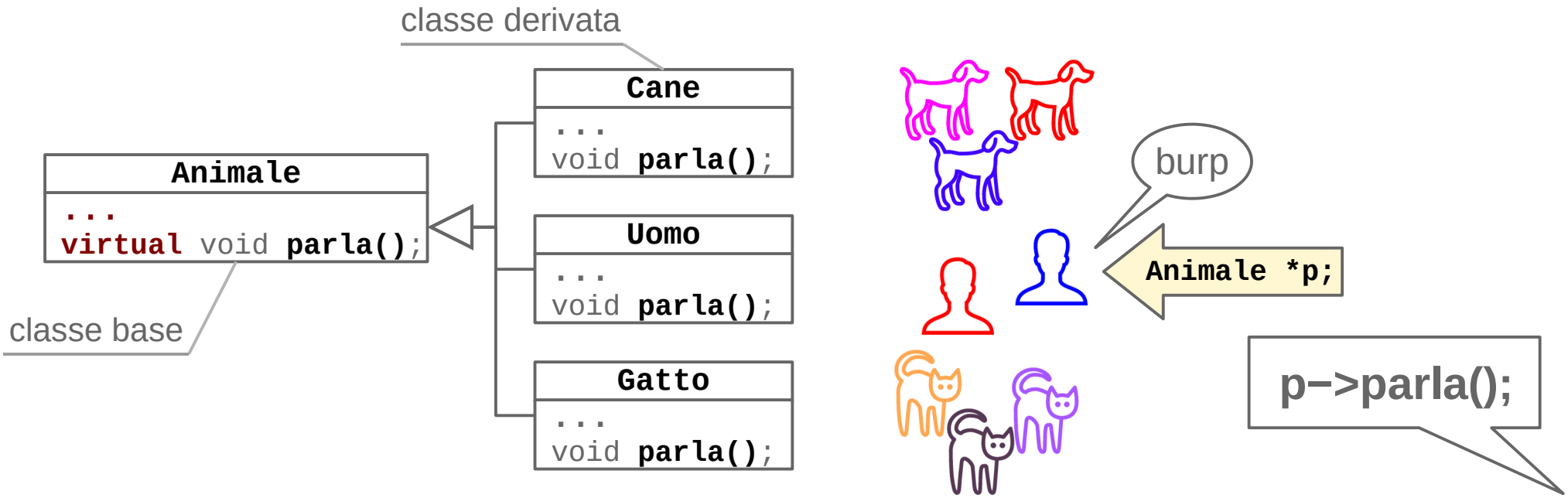
Polimorfismo run-time

- In altre parole, nonostante p sia un puntatore di tipo classe base (Animale)
  - Il sistema chiama la versione del metodo parla() implementata nella classe derivata (Cane, Gatto, ...)
  - Cioè **riconosce** l'oggetto puntato come appartenente ad una classe derivata



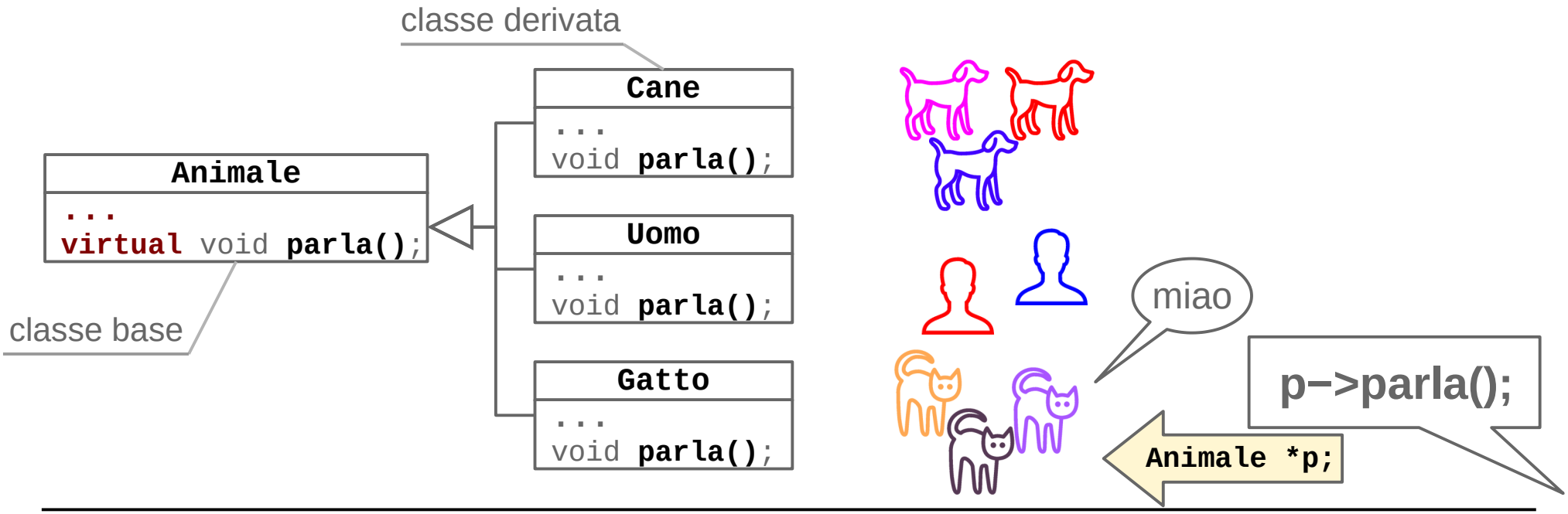
Polimorfismo run-time

- In altre parole, nonostante p sia un puntatore di tipo classe base (Animale)
  - Il sistema chiama la versione del metodo parla() implementata nella classe derivata (Cane, Gatto, ...)
  - Cioè **riconosce** l'oggetto puntato come appartenente ad una classe derivata



Polimorfismo run-time

- In altre parole, nonostante p sia un puntatore di tipo classe base (Animale)
  - Il sistema chiama la versione del metodo parla() implementata nella classe derivata (Cane, Gatto, ...)
  - Cioè **riconosce** l'oggetto puntato come appartenente ad una classe derivata



Esempio

```
#include <iostream>
// Classe base con metodo virtual: parla()
class Animale
{
 public:
  int funzioneACaso();
  virtual void parla()
  { std::cout << "???" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Cane : public Animale
{
 public:
  float Booh();
  virtual void parla()
  { std::cout << "Bau" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Gatto : public Animale
{
 public:
  virtual void parla()
  { std::cout << "Miao" << std::endl;}
};
... prosegue a destra
```

```
... segue da sinistra
// Eredita da Animale ma, per es,
// non ridefinisce parla()
class Uomo : public Animale
{
 public:
  bool cheNeSo();
};

int main()
{
  Cane  c;
  Gatto m;
  Uomo  u;
  // Puntatore a classe base
  Animale *p;
  // p punta a c
  p = &c;
  p->parla(); // Stampa "Bau"
  // p punta a u
  p = &u;
  p->parla(); // Stampa "???" xké
  // non e' ridefinito in Uomo
  return 0;
};
```

esPolimorfismoRunTime.cxx

Esempio

```
#include <iostream>
// Classe base con metodo virtual: parla()
class Animale
{
 public:
  int funzioneACaso();
  virtual void parla()
  { std::cout << "???" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Cane : public Animale
{
 public:
  float Booh();
  virtual void parla()
  { std::cout << "Bau" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Gatto : public Animale
{
 public:
  virtual void parla()
  { std::cout << "Miao" << std::endl;}
};
... prosegue a destra
```

```
... segue da sinistra
// Eredita da Animale ma, per es,
// non ridefinisce parla()
class Uomo : public Animale
{
 public:
  bool cheNeSo();
};

int main()
{
  Cane  c;
  Gatto m;
  Uomo  u;

  // Puntatore a classe base
  Animale *p;

  // p punta a c
  p = &c;
  p->parla(); // Stampa "Bau"

  // p punta a u
  p = &u;
  p->parla(); // Stampa "???" xké
  // non e' ridefinito in Uomo

  return 0;
};
```

esPolimorfismoRunTime.cxx

Esempio

```
#include <iostream>
// Classe base con metodo virtual: parla()
class Animale
{
 public:
  int funzioneACaso();
  virtual void parla()
  { std::cout << "???" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Cane : public Animale
{
 public:
  float Booh();
  virtual void parla()
  { std::cout << "Bau" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Gatto : public Animale
{
 public:
  virtual void parla()
  { std::cout << "Miao" << std::endl;}
};
... prosegue a destra
```

```
... segue da sinistra
// Eredita da Animale ma, per es,
// non ridefinisce parla()
class Uomo : public Animale
{
 public:
  bool cheNeSo();
};

int main()
{
  Cane  c;
  Gatto m;
  Uomo  u;
  // Puntatore a classe base
  Animale *p;
  // p punta a c
  p = &c;
  p->parla(); // Stampa "Bau"
  // p punta a u
  p = &u;
  p->parla(); // Stampa "???" xké
  // non e' ridefinito in Uomo
  return 0;
};
```

esPolimorfismoRunTime.cxx

Esempio

```
#include <iostream>
// Classe base con metodo virtual: parla()
class Animale
{
 public:
  int funzioneACaso();
  virtual void parla()
  { std::cout << "???" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Cane : public Animale
{
 public:
  float Booh();
  virtual void parla()
  { std::cout << "Bau" << std::endl;}
};
// Eredita da Animale e ridefinisce parla()
class Gatto : public Animale
{
 public:
  virtual void parla()
  { std::cout << "Miao" << std::endl;}
};
... prosegue a destra
```

```
... segue da sinistra
// Eredita da Animale ma, per es,
// non ridefinisce parla()
class Uomo : public Animale
{
 public:
  bool cheNeSo();
};

int main()
{
  Cane  c;
  Gatto m;
  Uomo  u;
  // Puntatore a classe base
  Animale *p;
  // p punta a c
  p = &c;
  p->parla(); // Stampa "Bau"
  // p punta a u
  p = &u;
  p->parla(); // Stampa "???" xké
  // non e' ridefinito in Uomo
  return 0;
};
```

esPolimorfismoRunTime.cxx

Esempio

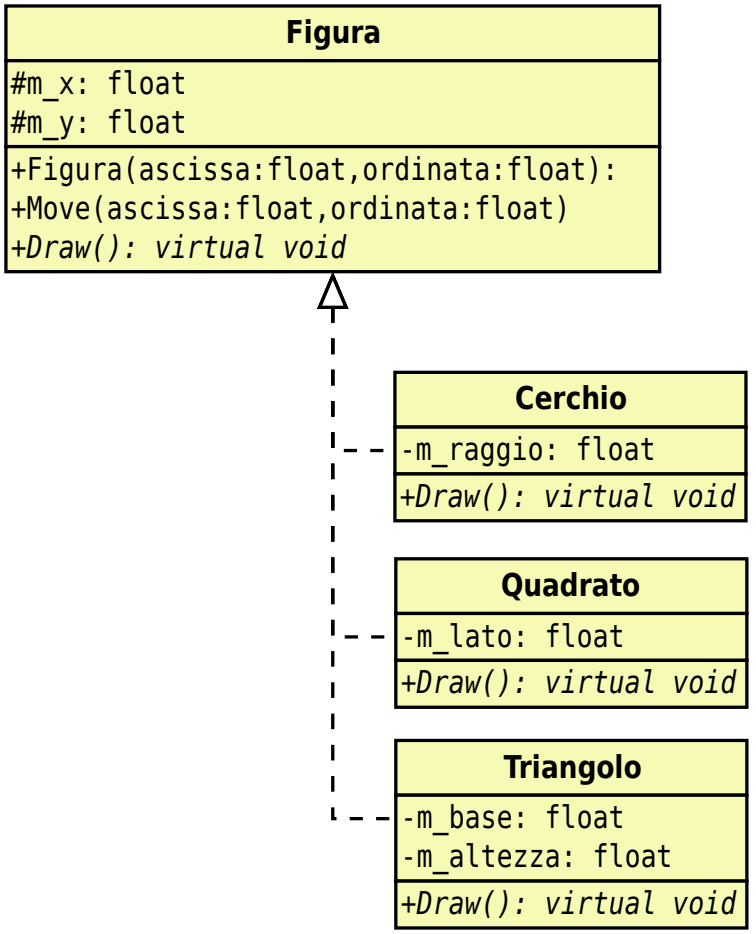
```
#include <iostream>

// Classe base con metodo virtual: parla()
class Animale
{
 public:
  int funzioneACaso();
  virtual void parla()
  { std::cout << "???" << std::endl;}
};

// Eredita da Animale e ridefinisce parla()
class Cane : public Animale
{
 public:
  float Booh();
  virtual void parla()
  { std::cout << "Bau" << std::endl;}
};

// Eredita da Animale e ridefinisce parla()
class Gatto : public Animale
{
 public:
  virtual void parla()
  { std::cout << "Miao" << std::endl;}
};

... prosegue a destra
```

```
... segue da sinistra

// Eredita da Animale ma, per es,
// non ridefinisce parla()
class Uomo : public Animale
{
 public:
  bool cheNeSo();
};

int main()
{
  Cane  c;
  Gatto m;
  Uomo  u;

  // Puntatore a classe base
  Animale *p;

  // p punta a c
  p = &c;
  p->parla(); // Stampa "Bau"

  // p punta a u
  p = &u;
  p->parla(); // Stampa "???" xké
  // non e' ridefinito in Uomo

  return 0;
};
```

esPolimorfismoRunTime.cxx

Es: programma di grafica

- Per esempio il polimorfismo run-time può essere utilizzato efficacemente nel realizzare un programma per disegnare figure geometriche
  - Come powerpoint
- In tal caso il **design** prevederà probabilmente una **classe base** (Figura) e varie **classi derivate**
  - una per ogni possibile figura: cerchio, triangolo, quadrato, ecc

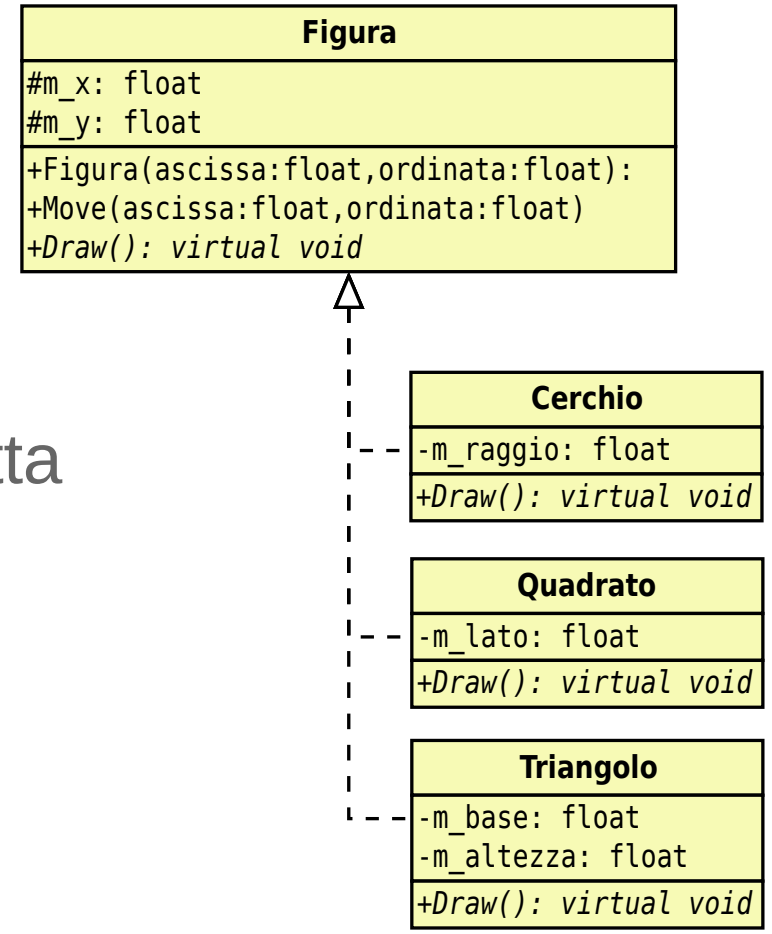


Es: programma di grafica

- La classe base avrà
  - Due dati membro per le coordinate: m_x e m_y
  - Metodi generici: per es. per spostare ∀ figura
  - Un metodo **virtuale** per disegnare la figura

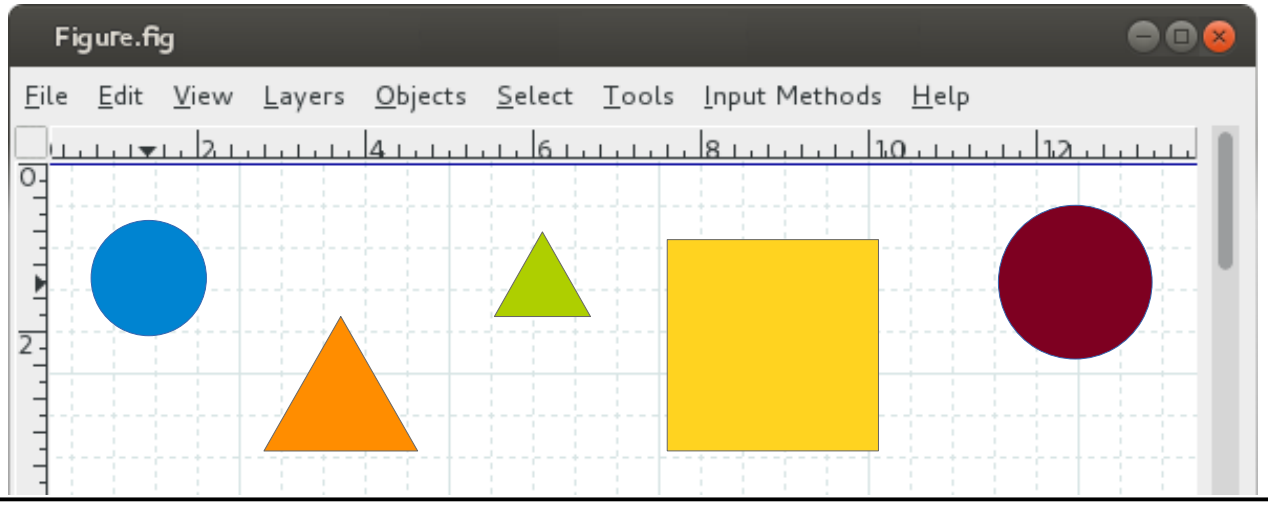
```
virtual void Draw();
```

- Ogni classe derivata avrà
  - la rappresentazione più adatta per la figura in questione: raggio, base e altezza, lato
  - l'implementazione **specifica** del metodo Draw(), cioè per disegnare quel tipo di figura



Es: programma di grafica

- Il design prevederà anche una classe `Schermo`
  - Per avere una “tela” su cui disegnare
- Nella rappresentazione di `Schermo` dovrà esserci un contenitore per ospitare le varie figure via via create dall'utente
  - Per esempio un container come std::vector

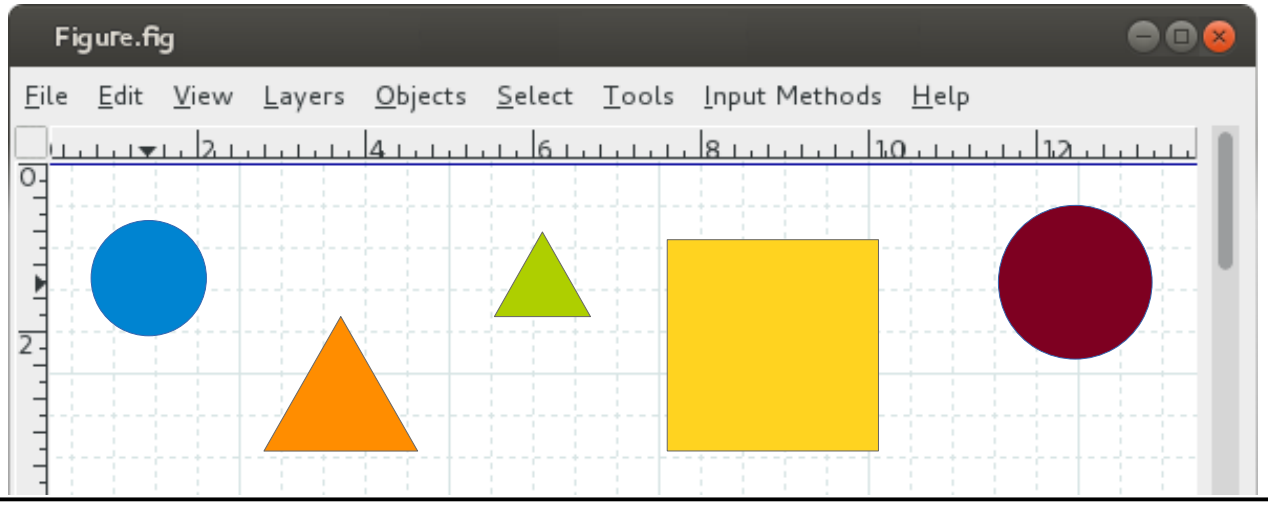


Es: programma di grafica

- Per sfruttare il polimorfismo run-time è necessario che il container ospiti gli oggetti creati, tramite **puntatori** alla classe **base** (Figura). Per es:

```
std::vector<Figura*> m_leFigu;
```

  - Ad ogni istante le figure da rappresentare saranno quelle contenute in quel momento in m_leFigu;



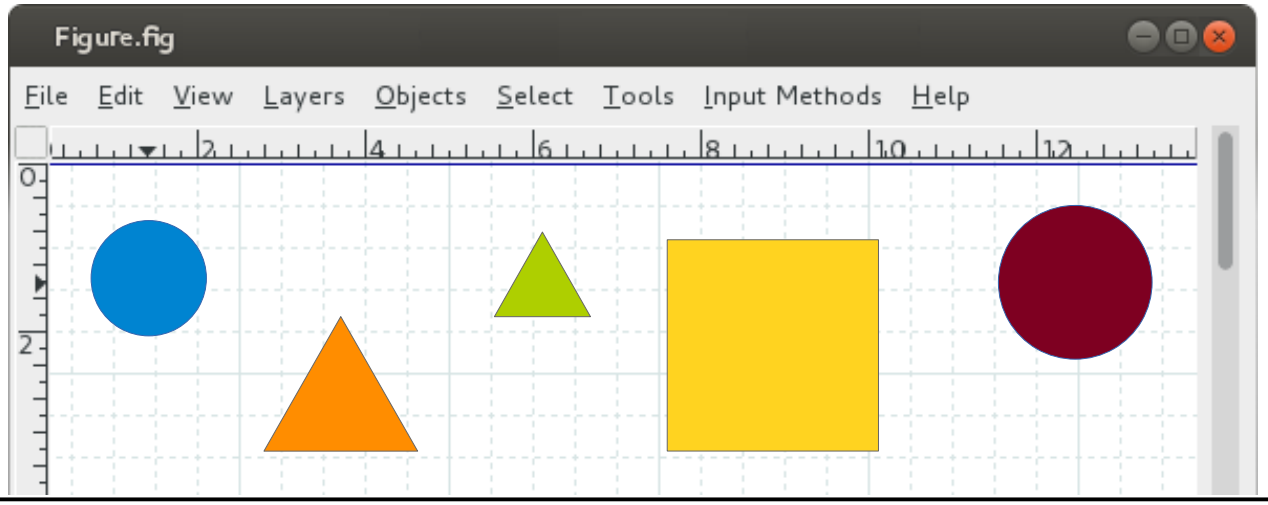
Es: programma di grafica

- Inoltre `Schermo` dovrà avere metodi di servizio
  - un metodo per aggiungere una nuova figura allo Schermo ogni qualvolta l'utente ne crea una. Es:

```
// Aggiunge figu a m_leFigu
void Add(Figura* figu);
```

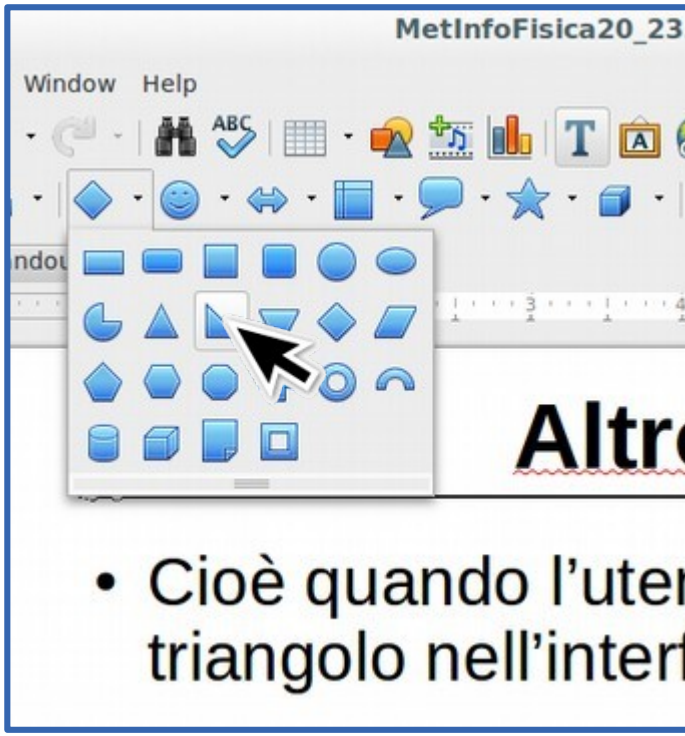
  - un metodo da chiamare quando lo `Schermo` va aggiornato: aggiunta o spostamento di una figura

```
void Aggiorna();  // Ridisegna lo schermo
```



Es: programma di grafica

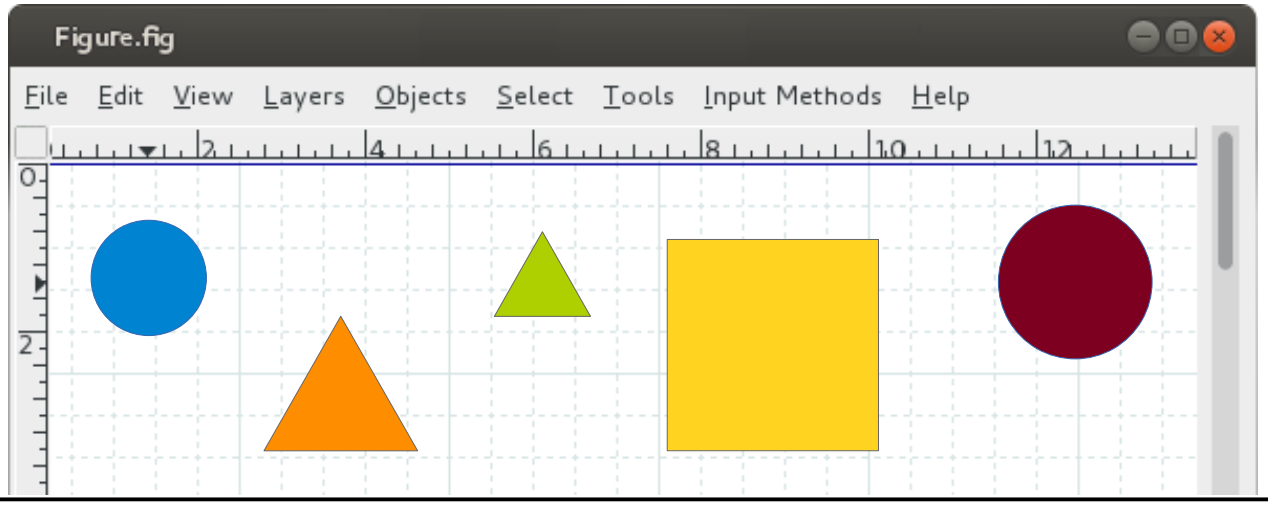
- Cioè quando l'utente pigia il tasto nuovo triangolo nell'interfaccia grafica
- Lo sviluppatore avrà fatto in modo che il codice
  - crei dinamicamente con `new` un nuovo oggetto della classe triangolo
  - crei un puntatore **`p`** di tipo classe **base** (Figura), che punti al nuovo oggetto della classe triangolo
  - venga chiamato `Schermo::add(p)` per aggiungere tale puntatore al container delle figure
  - venga chiamato `Schermo::aggiorna()` per aggiornare il contento della finestra



Es: programma di grafica

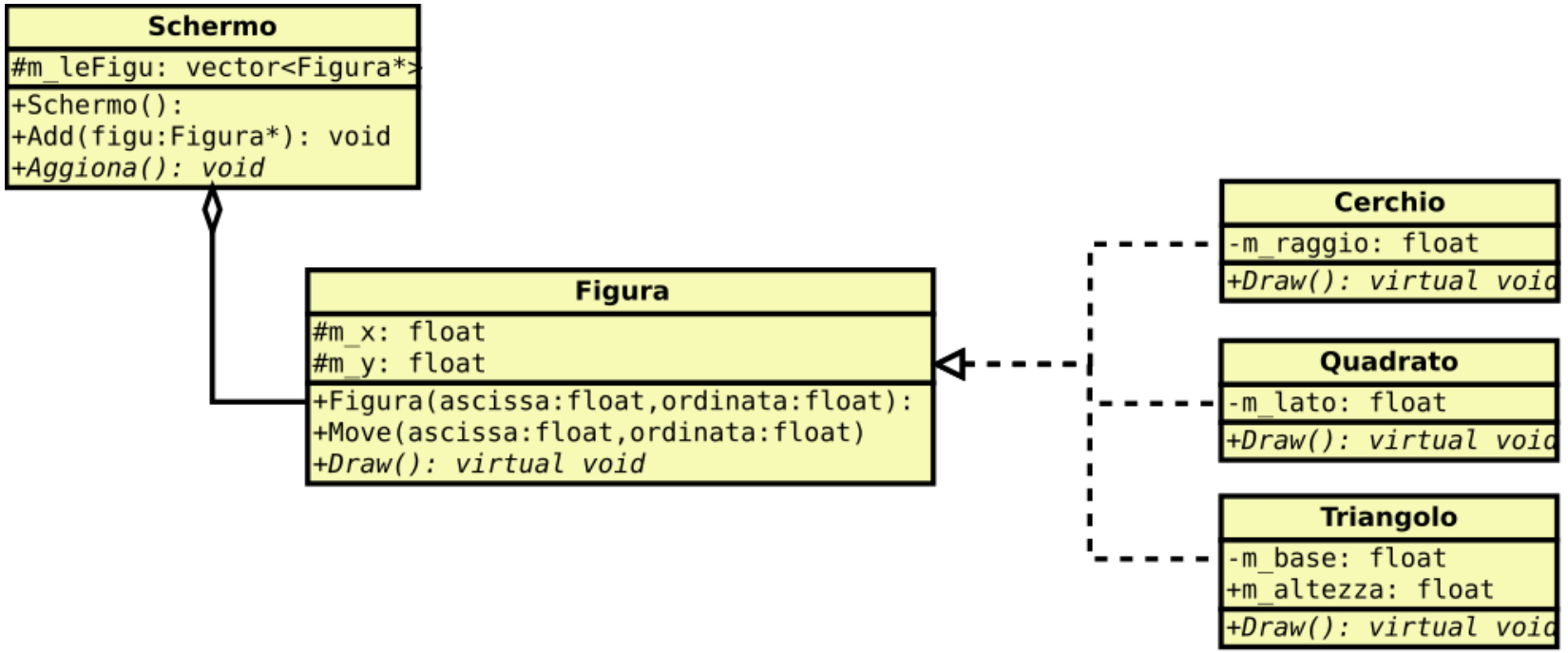
- Aggiorna() sarà un loop sul contenuto di m_leFigu
  - che contiene puntatori di tipo base (Figura) ad oggetti di tipo derivato (es: Cerchio) esistenti in quel momento
  - Chiamando il metodo virtuale Draw() tramite un puntatore alla classe base, il polimorfismo runtime **risolve** il tipo dell'oggetto puntato e chiama la versione corretta di Draw()

```
// Loop sulle figure definite al momento
for( int i=0; i < m_leFigu.size(); i++)
  m_leFigu[i]->Draw(); // Disegna la figura i-esima
```



Es: programma di grafica

- Riassunto del **design**
  - Classe **`Schermo`** che contiene vector di Figura* e è responsabile di gestire lo grafica
  - Classe base **`Figura`** con metodi virtuali
  - Una serie (estendibile) di classi derivate da Figura



Note Finali

- La peculiarità del polimorfismo runtime **non** è la possibilità di fare l'overloading nelle classi derivate
  - Quello si può fare sempre, a meno che il metodo non sia private delle classe base
- Ma la possibilità di accedere a metodi delle classi derivate tramite **puntatori** alla classe **base**
  - Quelli definiti **virtual** nella classe base
- Nel caso il metodo non fosse definito virtual
  - verrà chiamato il metodo della classe base
  - anche se l'oggetto puntato è della classe derivata

Note Finali

- Se un metodo è definito virtual la classe derivata è **libera** di fare l'overloading **o meno** tale metodo

```
virtual void funzione();
```

  - Se non è ridefinito, viene eseguito quello della classe base; esattamente come i metodi "normali"

- Se la dichiarazione termina con "**=0**" il metodo è "**pure virtual**": le classi derivate sono **obbligate** a ridefinirlo

```
virtual void funzione() = 0;
```

  - Se non lo fanno, il compilatore genera un errore

- Una classe base che ha solo metodi pure virtual è detta **classe astratta**
  - Utilizzate per definire **interfacce** generiche ed obbligare gli sviluppatori delle classi derivate ad attenersi allo schema

Riassunto

- Il **polimorfismo** run-time
  - Permette al sistema di riconoscere **run-time** il tipo dell'oggetto puntato da un puntatore alla classe base
- In questo modo non è necessario conoscere a compile time il tipo degli oggetti che verranno creati durante l'esecuzione del programma
  - Es: a priori non si sa cosa farà l'utente
- E' ovviamente possibile fare un design con un'implementazione specifica per ogni tipo
  - Ma in questo modo il codice sarebbe meno generico
  - Conterrebbe pericolose duplicazioni
  - Non sarebbe automaticamente estendibile a nuovi tipi